

Build and publish your first
Mac app for OS X Lion



Beginning OS X Lion Apps Development

Michael Privat | Robert Warner

Apress®

Beginning OS X Lion Apps Development



Michael Privat
Robert Warner

Apress®

Beginning OS X Lion Apps Development

Copyright © 2011 by Michael Privat and Robert Warner

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): 978-1-4302-3720-4

ISBN-13 (electronic): 978-1-4302-3721-1

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

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

President and Publisher: Paul Manning

Lead Editors: Steve Anglin and Michelle Lowman

Development Editors: Douglas Pundick and James Markham

Technical Reviewer: James Bucanek

Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Morgan Ertel,

Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman,

James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick,

Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh

Coordinating Editor: Jennifer L. Blackwell

Copy Editor: Kim Wimpsett

Compositor: MacPS, LLC

Indexer: SPi Global

Artist: SPi Global

Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media, LLC., 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 e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

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.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to <http://www.apress.com/source-code/>.

To my loving wife, Kelly, and our children, Matthieu and Chloé.

—Michael Privat

*To my beautiful wife, Sherry, and our wonderful children,
Tyson, Jacob, Mallory, Camie, and Leila.*

—Rob Warner

Contents at a Glance

Contents.....	v
About the Authors.....	x
About the Technical Reviewer	xi
Acknowledgments	xii
Introduction	xiv
■ Chapter 1: Starting to Build a Graphing Calculator	1
■ Chapter 2: Laying Out the User Interface	37
■ Chapter 3: Handling User Input	73
■ Chapter 4: Pimp My UI.....	119
■ Chapter 5: User Preferences and the File System	183
■ Chapter 6: Using Core Data.....	217
■ Chapter 7: Integrating Graphique into the Mac OS X Desktop.....	247
■ Chapter 8: Creating Help	289
■ Chapter 9: Printing	307
■ Chapter 10: Submitting to the Mac App Store.....	323
Index.....	365

Contents

Contents at a Glance	iv
About the Authors	x
About the Technical Reviewer	xi
Acknowledgments	xii
Introduction	xiv
■ Chapter 1: Starting to Build a Graphing Calculator	1
Using the Xcode Development Tools	2
Obtaining Xcode	2
Installing Xcode	3
Understanding Xcode	12
The Editor Area and the Jump Bar	16
The Navigator Area and the Navigator Selector Bar	17
Creating a Project	19
Understanding the Major Components	23
The Project and Targets	24
The Application Architecture	27
The Source Code and Resources	28
The About Dialog	34
Summary	35
■ Chapter 2: Laying Out the User Interface	37
Creating the Split View	38
Creating the Horizontal NSSplitView	39
Creating the Vertical NSSplitView	43
Creating the Equation Entry Panel	45
Using NSViewController	46
Laying Out the Custom Equation Entry Component	48
A Primer on Automatic Reference Counting	49
Using IBOutlet	50
Hooking Up the New Component to the Application	51
Resizing the Views Automatically	53
Further Customizing the Components	56
Creating the Graph Panel	57

Adding the Horizontal Slider	58
Adding the Table View	60
Adding the Graph Panel to the Application	61
Creating the Table of Recently Used Equations	62
Creating the Data Source	67
Displaying the Data	69
Adding a Toolbar	71
Summary	72
■ Chapter 3: Handling User Input	73
Resizing the Views	74
Resizing the Window	74
Constraining the Split View Sizes	76
Constraining the Window Size	79
Collapsing a Subview	83
Handling Button Presses	85
The Model-View-Controller Pattern	85
Using IBAction	85
Creating the Model: Equation	87
Communication Among Controllers	90
Validating Fields	91
Validating After Submitting	91
A Better Way: Real-Time Validation	100
Graphing the Data	104
Calculating and Caching the Data	106
Talking to the Table: Outlets and Delegates	108
Changing the Interval in the Domain	112
Using Key-Value Coding	113
Binding the Value to the Slider	114
Summary	117
■ Chapter 4: Pimp My UI	119
Creating a Graph View	120
Creating a Custom View	120
Linking the New Custom View to the Controller	121
Plotting the Graph	124
Toggling Between Text and Graph	129
Adding the Tab View	130
Adding the Views to the Tabs	132
Switching the Controller to the Tab View	134
Creating a Smarter Equation Editor	136
Adding Attributes to the Equation Entry Field	137
Creating the Tokens	138
Parsing the Equation	140
Implementing the Method to Tokenize the Equation	142
Detecting Multiple Decimal Points	161
Testing the Tokenizer	164
Showing the Equation	169
Updating the Validator	175

Updating the Evaluator.....	177
Summary	182
Chapter 5: User Preferences and the File System	183
Managing User Preferences.....	184
Understanding NSUserDefaults.....	186
Setting the Font for the Equation Entry Field	188
Setting the Line Color	196
Creating a Custom Preferences Panel	201
Using the Local File System.....	208
Browsing the File System	209
Writing to the File System.....	210
Reading from the File System.....	211
Exporting Graphs as Images	211
Summary	216
Chapter 6: Using Core Data.....	217
Stepping Up to Core Data.....	218
Adding Core Data to the Graphique Application.....	218
Adding the Core Data Framework.....	218
Creating a Data Model	221
Designing the Data Model.....	222
Initialize the Managed Object Context	229
Storing Recently Used Equations	232
Querying the Persistent Store to Get the Group Entity	234
Creating the Equation Managed Object and Adding It to the Persistent Store.....	235
Committing	235
Putting Everything Together into the Final Method.....	236
Reloading Recently Used Equations	237
Tightening the Control over the Outline View	243
Using NSOutlineViewDelegate	243
Handling Equations Selection	244
Preventing Double-Clicks from Editing.....	245
Summary	246
Chapter 7: Integrating Graphique into the Mac OS X Desktop.....	247
Dealing with Graphique XML Files	247
Producing a Graphique File	247
Loading a Graphique File into the Application	252
Registering File Types with Lion.....	253
Defining the New UTI for the .graphique Extension	253
Registering Graphique as an Editor for Graphique Files	255
Handling Graphique Equation Files	257
Using Quick Look to Generate Previews and Thumbnails.....	259
Creating the Quick Look Plug-in	261
Implementing the Preview	264
Testing the Plug-in.....	267
Implementing the Thumbnail	271
Distributing the Quick Look Plug-in with the Graphique Application.....	275

Adding an Item to the Menu Bar	277
Understanding NSStatusBar and NSStatusItem.....	277
Adding a Status Item to Graphique	278
Building the Status Item Menu	279
Integrating the Status Item	282
Heeding Apple's Advice Regarding Menu Bar Icons	284
Summary	288
Chapter 8: Creating Help	289
A Word on Help	290
Understanding Help Books.....	290
Creating Your Help Book	291
Creating the Directory Structure	291
Creating the Main Help File.....	292
Creating the Rest of Your Help Files	294
Creating the Help Index.....	297
Setting Up Your Plist File	297
Importing Your Help Book into Your Xcode Project.....	300
Updating Your Application's Plist File	301
Viewing the Help	302
Bookmarking a Page.....	303
Performing a Search.....	304
Summary	306
Chapter 9: Printing	307
Printing the Graph View	307
Printing the Graph.....	307
Drawing for the Printer	316
Spanning to Multiple Pages	318
Calculating the Number of Pages	319
Determining the Page Size	320
Drawing the Page	321
Summary	322
Chapter 10: Submitting to the Mac App Store.....	323
Reviewing the Guidelines	323
Finishing the App	324
Terminating Graphique When Its Window Closes	324
Adding the Icon.....	324
Reviewing the Property List File	327
Cleaning Up the Menu.....	328
Setting the Initial Window Size and Location.....	331
Signing the Code.....	334
Sandboxing the App.....	349
Building for Release.....	352
Setting Up Your Web Site.....	353
Using the Artwork	354
Creating the Web Site	354

Submitting the App356

 Setting Up Your iTunes Connect Account356

 Uploading Your Application357

Summary363

Index 365

About the Authors



Michael Privat is the president and CEO of Majorspot, Inc., developer of several iPhone and iPad apps:

- Ghostwriter Notes
- My Spending
- iBudget
- Chess Puzzle Challenge

He is also an expert developer and technical lead for Availity, LLC, based in Jacksonville, Florida. He earned his master's degree in computer science from the University of Nice in Nice, France. He moved to the United States to develop software in artificial intelligence at the Massachusetts Institute of Technology. Coauthor of *Pro Core Data for iOS* (Apress, 2011), he now lives in Jacksonville, Florida, with his wife, Kelly, and their two children.



Rob Warner is a senior technical staff member for Availity, LLC, based in Jacksonville, Florida, where he works with various teams and technologies to deliver solutions in the health-care sector. He coauthored *Pro Core Data for iOS* (Apress, 2011) and *The Definitive Guide to SWT and JFace* (Apress, 2004), and he blogs at www.grailbox.com. He earned his bachelor's degree in English from Brigham Young University in Provo, Utah. He lives in Jacksonville, Florida, with his wife, Sherry, and their five children.

About the Technical Reviewer



James Bucanek has spent the past 30 years programming and developing microcomputer systems. He has experience with a broad range of technologies, from embedded consumer products to industrial robotics. James is currently focused on Macintosh and iPhone software development. When not programming, James indulges in his love of the arts. He earned an associate's degree from the Royal Academy of Dance in classical ballet and occasionally teaches at Adams Ballet Academy.

Acknowledgments

When I was first presented with the opportunity to coauthor this book with Rob Warner, I was both excited and nervous. I was excited because our previous book on Core Data has been a success and Rob is a great partner to work with. But I was nervous because of the time investment we would both have to pour into this. In the end, we pulled and pushed each other to the finish line to produce this new book, and, once again, it was all worth it.

My wife, Kelly, and my children, Matthieu and Chloé, have thankfully been very supportive throughout the project and helped me with words of motivation and a constant flow of coffee. Matthieu and Chloé think it's cool that their dad wrote a book, and I'm all about impressing my kids while I still can.

As always, the folks at Availity with whom I spend my daylight time have all been encouraging to Rob and me in this effort. Availity is a company that promotes continuous professional self-improvement, and this is yet another example. Among plenty of other folks, Trent Gavazzi, Ben Van Maanen, and Steve Vaughn have all stopped by with words of encouragements that went a long way when I felt overwhelmed between work and writing.

Lastly, it's only fair to acknowledge the Apress team for their constant support and for providing resources to help with the completion of the project. The technical reviewer they hired to work with us, James Bucanek, was a wealth and knowledge. James, I apologize for that time I made you spit your coffee all over your monitor with my writing....

Michael Privat

Although writing can be easy, writing well never is, at least for me. I agonize over word choices, sentence constructs, and flow. Never does writing become more difficult, however, than in this section, as we try to thank those who helped create this book, whether directly or indirectly. The danger of omitting someone who merits thanks lurks behind every paragraph, and the peril of sounding trite or rote looms in every sentence. If you've been overlooked or unfairly treated, I blame the editors!

Actually, working with the people at Apress has been another great adventure. From the beginning ideas for this book to its conclusion, we've been supported and challenged to greatness. Steve Anglin always has great ideas and doesn't settle for almost. Jennifer Blackwell was both patient and pleasant as we worked through issues. Going through my e-mail reminds me of more key folks who provided direction or assisted: Michelle Lowman, Debra Kelly, Douglas Pundick, James Markham, and Kim Wimpsett. Our technical reviewer, James Bucanek, was both fastidious and amazing, keeping us on our toes, steering us right, and helping us solve some technical challenges. Thanks, James!

Thank you, Sherry, for letting me chase my dreams.

Juggling two book projects while holding down a full-time job and raising a family can cut into one's sleep schedule. Tyson has now dubbed me "Sleepyhead Codesauce," and my other children roll their eyes at my "nerd books" and "nerd meetings." Thanks, Tyson, Jacob, Mallory, Camie, and Leila for letting me have my fun and for putting up with my crankiness.

Conning Michael Privat into writing yet another book with me was somewhat devious—I've learned he can't back down from a challenge. Thanks, Michael, for climbing on board one more time. This book was another great ride, and I'd have been stuck without you. Betcha can't write one on Dart.

Working my day job at Availity provides me opportunities to learn and grow from some amazing people. Once again, I thank all of them for their support and interest, particularly Trent Gavazzi, Jon McBride, and the rest of the senior team. As more and more Availity people get Macs, I expect to see Graphique on a lot of desktops soon.

Finally, I thank my parents and my siblings for the love of learning and writing that we share. Thanks for your support and your interest.

Rob Warner

Introduction

Mac OS X offers an amazing development environment for scores of technologies. It seems that developers from numerous camps are migrating to Mac en masse. Scan the room at any Ruby or Rails conference, for example, and you'll see programmers coding on Macs almost exclusively. As developers move to Mac, almost inevitably they eventually discover the itch to write native tools on the Mac platform. We've written this book to help those of you who hear the siren call of Xcode and just need a little guidance through the rocky waters of Objective-C and Cocoa to be able to develop your own apps.

The Premise

We based this book on a simple premise: the best way to learn to program in a new language or environment is to build something real, something finished, something that's actually useful. Too many books on programming leave out things like error handling or help files or other programming topics that might be considered on the periphery of teaching programming principles or language topics so they can focus on the core of what they're trying to teach. While this focus can be useful, it leaves a gap between building an almost-product and building a finished product, leaving readers to seek information elsewhere for crossing that chasm. Our pledge is to bridge that gap in this book. By taking you from project creation through adding features to finishing the project to publishing it in the Mac App Store, we hope to give you the information you need to finish projects, not just start them.

It's been an interesting premise. It's brought challenges and discussions about what to include, what to leave out, and how best to illustrate certain aspects of Mac OS X development. As we've developed this application, assumptions we made early about topics that would be essential proved trivial, and topics we deemed unimportant reared up as integral to the product. We've had to adjust as we went, including as many topics as made sense in the context of the app, a graphing calculator called Graphique.

Developing a complete application as part of a beginning book presented its own challenges as well. Getting a real application to "done" often involves some deeper topics or code that doesn't seem strictly a beginner's domain. We've tried to balance completeness, correctness, and simplicity in the code base for the application we develop and hope we've succeeded.

We believe the final app, available in the Mac App Store, is useful. We also believe that its source code and the text of this book can help you learn to develop complete apps fit for publishing.

The Audience

You have a Mac, and you want to develop apps for it. That's our audience. We target more specifically folks who want to publish those apps in the Mac App Store and devote a detailed

chapter to doing so. You can ignore that chapter if you don't want to publish through the Mac App Store.

This is a beginner's book, but we don't teach the basics of programming here, so if you don't know what a for loop is or shy from a command prompt, you might struggle. We do teach the basics of programming in Objective-C and Cocoa for Mac OS X, so if you understand programming basics, even if you know nothing about programming for the Mac, you'll do fine in this book. Advanced programmers moving from other languages or platforms will find this book useful, as will iOS developers looking to leverage their Cocoa skills to the Mac OS X platform.

How This Book Is Organized

This book begins with preparing your computer to build OS X Lion apps by walking you through the installation of Xcode, the development tool for building OS X apps. You then create the project for Graphique, the graphing calculator you build throughout this book. By the time you finish the first chapter, you'll be able to build and run the fledgling Graphique.

Each subsequent chapter adds more functionality to Graphique, building on the project from the previous chapter. The downloadable source code captures the state of the project at the end of each chapter, so if you insist on skipping a chapter, you can. To get the most from the book, though, work through it from the first page to the last, typing in the code, building the project, and running it as you go. After your initial read, keep this book handy for reference, so you can jump to specific topics as you build your own applications.

The final chapter walks you through the sometimes-confusing process of submitting your app to the Mac App Store. Though you won't submit your copy of Graphique, since we already submitted the app and it's already available, you'll follow the same steps when you submit your own applications.

Source Code and Errata

The complete source code for Graphique, divided into chapters, can be downloaded from the Apress web site at www.apress.com. Download it, learn from it, and use it in your own applications. As we uncover bugs or typos in the book or code, we'll update the errata section on the Apress web site.

How to Contact Us

We'd love to hear from you, whether it's questions, concerns, better ways of doing things, or triumphant announcements of your apps landing on the Mac App Store. You can find us here:

Michael Privat:

E-mail: mprivat@mac.com

Twitter: [@michaelprivat](https://twitter.com/michaelprivat)

Blog: <http://michaelprivat.com>

Rob Warner:

E-mail: rwarnier@grailbox.com

Twitter: [@hoop33](https://twitter.com/hoop33)

Blog: <http://grailbox.com>

Starting to Build a Graphing Calculator

When Apple announced Mac OS X 10.7, also known as Lion, it did so under the banner “Back to the Mac.” Mac OS X Lion incorporates lessons Apple has learned from the wildly successful iOS platform, available on iPhones, iPads, and iPod touch devices, and the excitement it has garnered presents lucrative opportunities for you, as a software developer, to create and distribute applications for this platform. Mac users consistently demonstrate passion for their computing platform, a willingness to pay for software, and a demand for quality and innovation. If your motivations don’t issue from financial fountains, developing Mac software provides you with a way to return the passion for computing, quality, and innovation to the Mac user community. This book gives you all the information you need to develop and distribute apps that tap into that passion.

In this book, we develop a graphing calculator called Graphique. The project develops as we go through the book, and each chapter adds more functionality to the application. At the conclusion of the book, the Graphique app is complete, and we show you how to submit to the Mac App Store. The feature set for Graphique has been selected not just to make sure Graphique graphs calculations in a superior way but also to demonstrate Mac application principles that you can learn from and apply to your own applications. This book works best when you read it in front of your Mac, typing in the code, running the application along the way, and making sure you understand each principle as you go. This book also works well as a reference, allowing you to jump to a specific topic to understand how to implement it in your own projects.

To submit apps you develop to the Mac App Store, you must enroll in the Mac Dev Center program, which costs \$99 a year. In addition to App Store submission privileges, your membership in the Mac Dev Center grants you the following:

- Access to prerelease versions of operating systems
- Access to prerelease versions of developer tools
- Technical support from Apple engineers

- Access to Apple developer forums
- Access to developer videos

To enroll in the Mac Dev Center, point your browser to <http://developer.apple.com>, click the link that says Mac Dev Center, and follow the instructions both to register as an Apple Developer, if you haven't already, and then to enroll in the Mac Dev Center. Apple will send you an e-mail when you are successfully enrolled, and you can then log in to the Mac Dev Center.

Using the Xcode Development Tools

The next section explains how to obtain and install Xcode, the tool Apple provides for developing Mac applications. If you've already installed Xcode or feel like you need no help obtaining and installing it, you can skip ahead to the section "Creating a Project."

Obtaining Xcode

Apple released Xcode 4 in March 2011 with a changed pricing structure. Joining the Mac Dev Center program (<https://developer.apple.com/devcenter/mac/index.action>) still costs \$99 a year, but the Xcode software had always been free for anyone to download, install, and use to develop, compile, and distribute applications. In the new pricing structure, Xcode 4 became available as a free web download only to paying members of the Mac Dev Center and the iOS Dev Center. Apple directed all others to its App Store to buy Xcode 4 for \$4.99. The web community reacted as if pricing had shot up to Visual Studio levels, demonstrating that the emotional gap between free and \$5 far exceeds that between, say, \$20 and \$25. When Xcode 4.1 appeared in conjunction with the release of Lion, however, Apple dropped the price back to free, quelling any lingering resentment.

To download Xcode 4 as a Mac Dev Center member, point your browser to the Mac Dev Center (<https://developer.apple.com/devcenter/mac/index.action>) and follow the link featured prominently on the Mac Dev Center home page to download the Xcode 4 disk image. Be prepared to wait a while, because the disk image is large, and Apple doesn't do patches (meaning that upgrading to future versions will entail downloading the entire new version, not just the changed bits). If you don't want to join the Mac Dev Center or you just like using the App Store, you can search the App Store for *Xcode* and click the link to download it.

However you obtain Xcode 4, you must install it. Proceed to the next section for step-by-step instructions on how to do so.

Installing Xcode

The steps for installing Xcode differ slightly depending on whether you downloaded it from the Web or through the App Store. We cover instructions for installing the web download first.

Installing from the Web Download

Xcode downloads from the Web as a disk image (.dmg) file. Depending on your browser and its settings, the disk image file may automatically open, or you may have to double-click it to open it. Either way, the disk image file mounts as a virtual drive and opens in a Finder window, as shown in Figure 1–1.



Figure 1–1. *The mounted Xcode disk image*

You can read notes about Xcode and installation instructions by opening the About Xcode file. To begin the installation, double-click the Xcode icon, denoted by an open brown box. The installer launches and tells you that it will determine whether Xcode can be installed on your system, as shown in Figure 1–2. Click Continue.

After your system passes the checks, you get another window telling you that the installer is going to guide you through the installation, as shown in Figure 1–3. Apple takes you slowly and interactively through the installation process, and you'll click a number of buttons before the installation completes. In this window, click Continue.

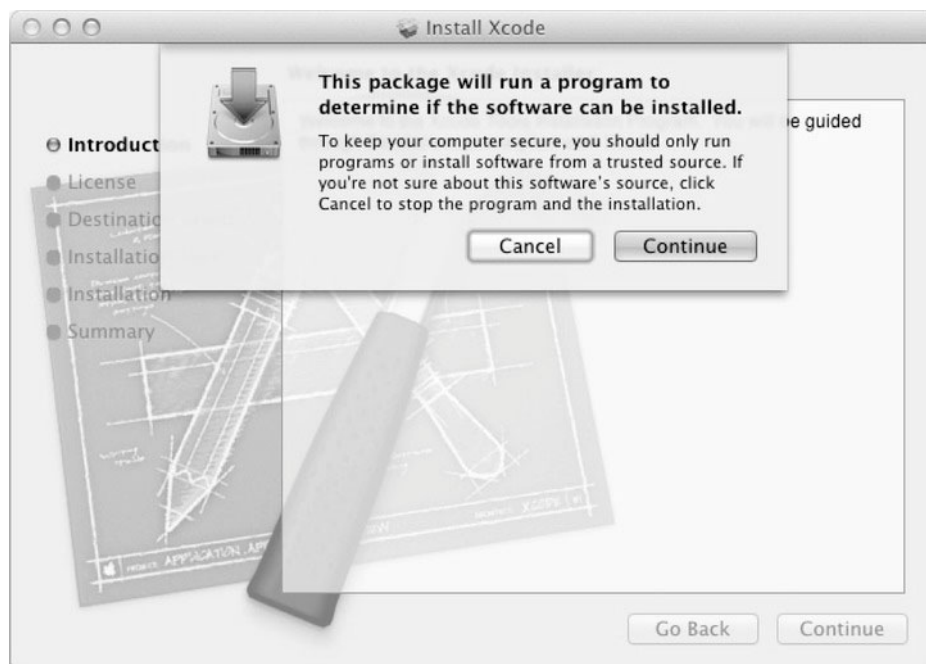


Figure 1–2. Xcode determining whether it can install

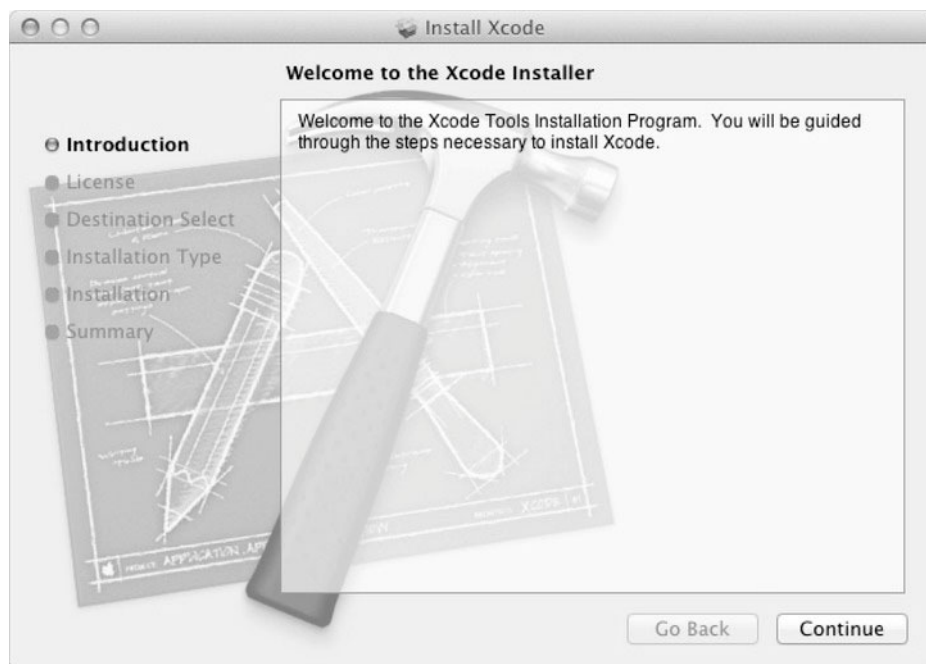


Figure 1–3. Xcode preparing to install

Anytime you install software, you must agree to things that you never read and, even if you did read them, you wouldn't understand. Our lawyers tell us to tell you to read all agreements carefully and consult with your lawyers before ever dreaming of clicking Continue. However you handle the next screen, shown in Figure 1-4, start to dismiss it by clicking Continue.



Figure 1-4. *The Xcode license agreement*

As soon as you click Continue, Apple asks you whether you actually agree to the terms you haven't read. Dismiss this message, shown in Figure 1-5, by studiously weighing all your options, by which we mean move your mouse pointer as fast as you can to the Agree button and click it.

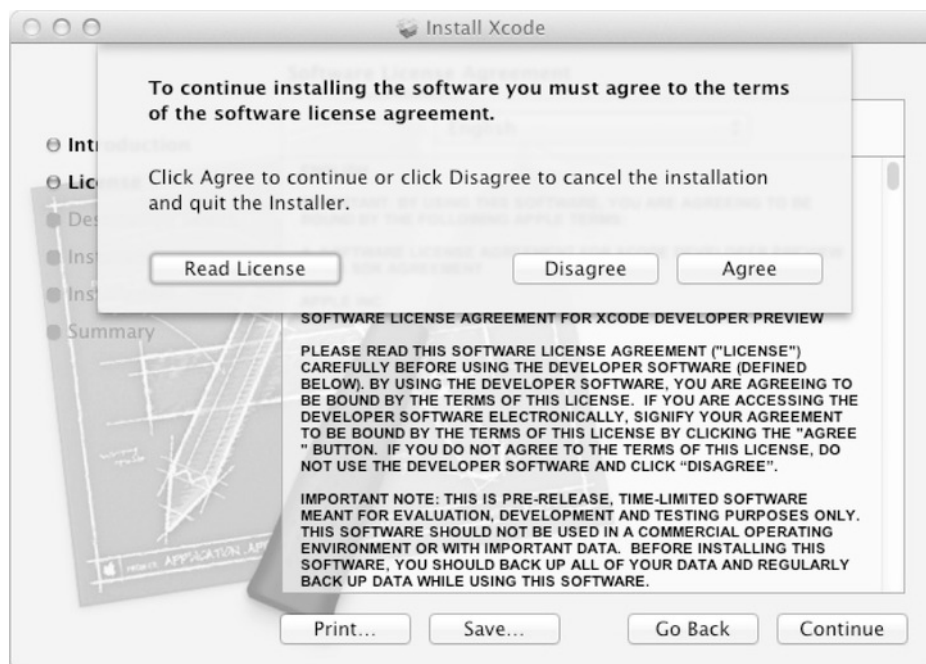


Figure 1-5. Responding to the Xcode license agreement

In the next dialog, shown in Figure 1-6, you have some choices. You can, for example, change the installation location from the default, which is `/Developer` (the `Developer` directory in the root of your boot drive). You probably should leave this at the default unless you have some reason to change it, such as that you want to run more than one version of Xcode on the same computer. The next option is whether to install the System Tools, which provide support for such tools as Instruments, the profiling tool, and Git, the version-control tool. You should install these. You should also install the UNIX Development tools, because many developer tools (such as various Ruby gems) you'll use over time will require that the UNIX Development tools be present so they can compile and install. Lastly, you can select to install or skip the documentation. We say to install it, because you'll count on it fiercely as you develop software, but it's your hard drive and your choice. If you don't install it, you'll need an active Internet connection any time you view documentation. After making your installation choices, click **Continue**.

Next, Apple offers you a **Change Install Location** button, as shown in Figure 1-7, to let you change the drive on which Xcode will install. If you feel compelled to relocate, do so, but then click the **Install** button to set the installation in motion. You'll be asked to type your password, as shown in Figure 1-8, so that the installer can write to privileged locations on disk. Type your password and click **Install Software**.

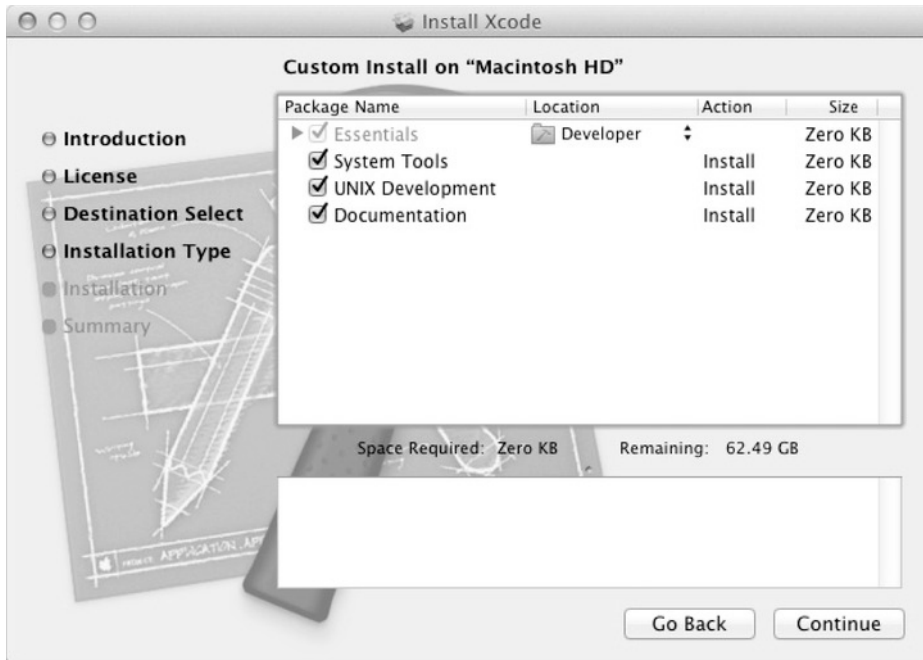


Figure 1–6. *Selecting Xcode installation options*

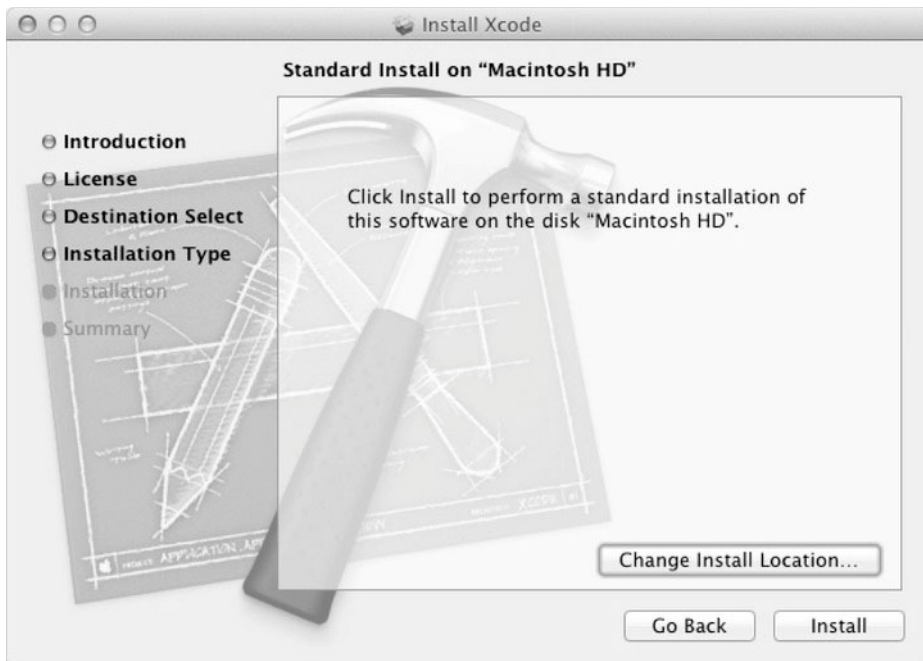


Figure 1–7. *The installer letting you change the install location*



Figure 1-8. *The installer prompting you for your password*

The next screen (shown in Figure 1-9) starts motoring through installation tasks, but it's going to take a while before Xcode is installed and ready to use. You'll be asked no more questions until the installation completes, though, so now you can step away for a bathroom break, to prepare your taxes, or to solve Fermat's Last Theorem in a novel way. When the installation finishes, it will gleefully chime and present the window shown in Figure 1-10. Click Close to dismiss it. Congratulations—you've installed Xcode and are ready to proceed to the section "Understanding Xcode."



Figure 1-9. *The installer writing the Xcode files to your drive*



Figure 1–10. *The completion of a successful installation*

Installing from the App Store

If you want to install Xcode from the App Store, launch the App Store application and either go to the Developer Tools section or search for Xcode. Install as you would with any other app from the App Store: click the Free button to turn it into an Install App button, and then click the Install App button. Enter your Apple ID and password in the dialog that appears, and click Sign In. The Launchpad opens and displays the Xcode icon in muted gray tones, with a progress bar and the word “Downloading...,” as shown in Figure 1–11. Depending on your connection speed, this download will take either a long time or a really long time. Go find something else to do while the download finishes.



Figure 1–11. *Xcode downloading in your Launchpad*

The App Store, unlike with other apps you install, doesn't install the Xcode app directly. Instead, when the download completes, the App Store installs an app into your /Applications directory called Install Xcode that, as the name suggests, is an installer for Xcode. It looks like Figure 1–12 in your Launchpad. Launch this app to begin installing Xcode.



Figure 1–12. *The Install Xcode app installed from the App Store*

The App Store version of Xcode offers a somewhat more streamlined installation process than the web-download version, using fewer screens that look a little different. When you launch the Install Xcode app, you see the splash screen shown in Figure 1–13. Yours may look a little different, depending on whether you already have a version of Xcode installed on your machine. Click Install to start the installation process.

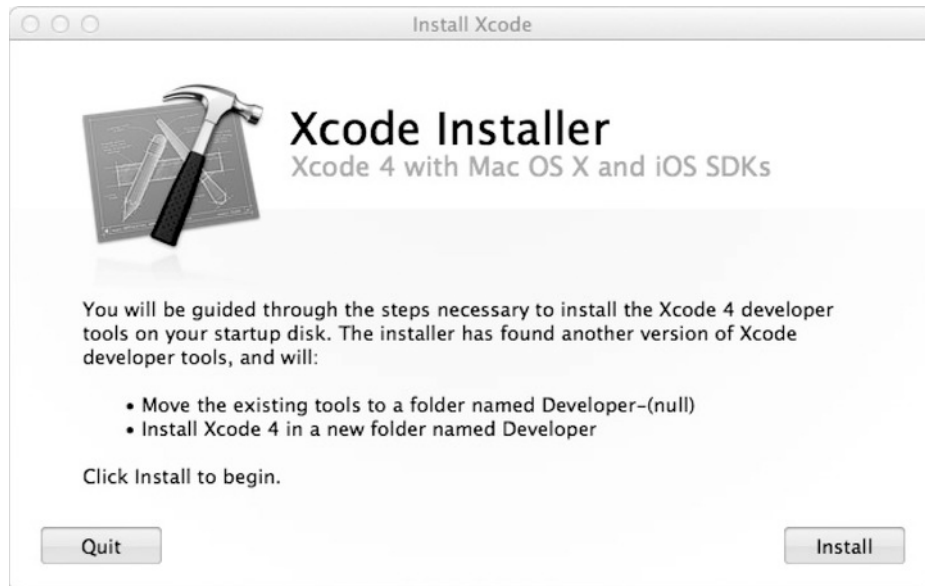


Figure 1–13. *The Xcode Installer splash screen*

The installer then shows the Xcode and the iOS SDK agreements, as shown in Figure 1–14. Click Agree to accept the agreements and proceed. You'll be prompted for your password so that the Xcode installer can install Xcode files to privileged areas on your drive, as shown in Figure 1–15. Type your password, click OK, and the installer starts copying files to your drive, as shown in Figure 1–16. When the installation completes, Xcode automatically launches.

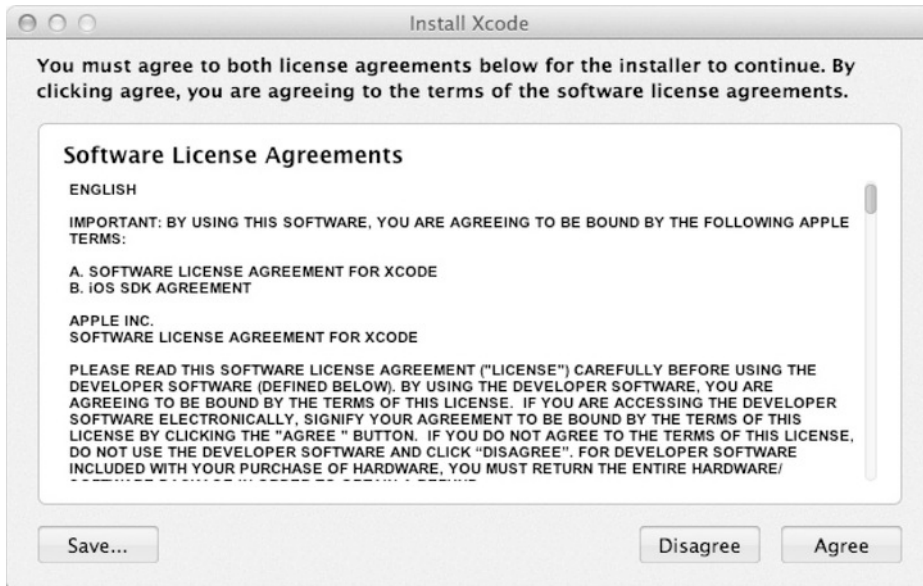


Figure 1–14. The agreements for Xcode and the iOS SDK



Figure 1–15. Entering your password so the installer can proceed



Figure 1–16. *The installer writing the Xcode files to your drive*

After you install Xcode from the App Store, you can reclaim about 3GB of disk space by backing up the Install Xcode app from /Applications, in case you ever want to reinstall without downloading, and deleting it from your drive.

Understanding Xcode

With Xcode safely installed, you're ready to launch it using your favorite launching method. When you launch Xcode, you see a splash screen, shown in Figure 1–17, that offers you the option to create a new Xcode project, connect to a source code repository, learn more about Xcode, open a browser to <http://developer.apple.com>, open one of your recent projects, or open some other project. This screen is usually helpful, but if you prefer, you can uncheck the box beside "Show this window when Xcode launches" to banish it from reappearing.



Figure 1–17. *The Xcode splash screen*

To acquaint yourself with the major sections of Xcode, proceed by clicking the “Create a new Xcode project” option. Xcode asks you to choose the template for your new project, so select Application under Mac OS X on the left and Command Line Tool on the right, as shown in Figure 1–18, and click Next. Xcode asks you for a product name and a type. For Product Name, type **HelloWorld**, for Company Identifier, type **book.macdev**, and for Type, select Foundation, as Figure 1–19 shows. Click Next. Xcode asks you where to create your project folder and offers you to create a local Git repository for the project. Select the parent directory for your project (something like Projects or Development in your home folder—wherever you normally store your coding projects), uncheck the box for creating a local Git repository for this project, and click Create. The Xcode integrated development environment (IDE) launches and opens your project, as shown in Figure 1–20.

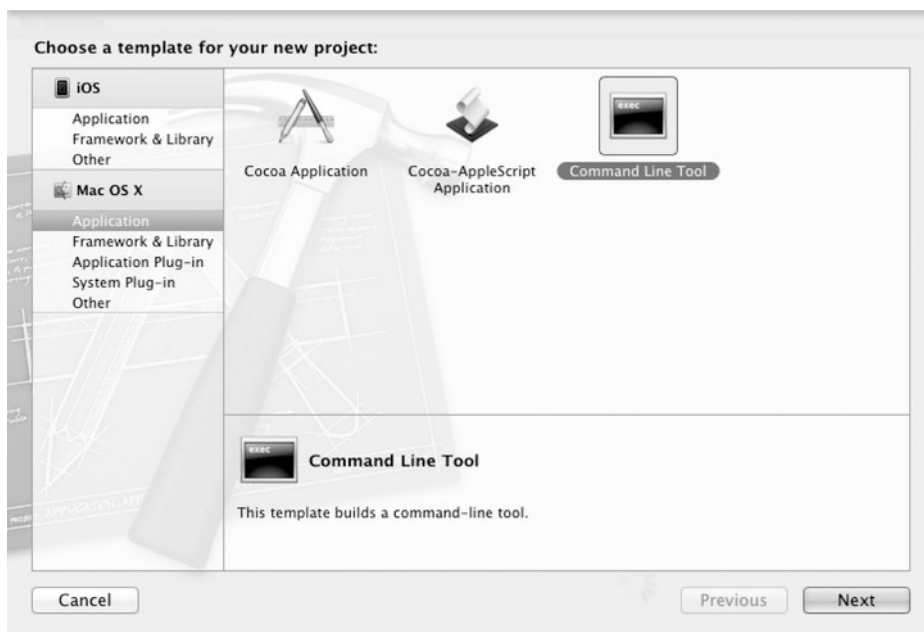


Figure 1-18. Choosing a template for your new project

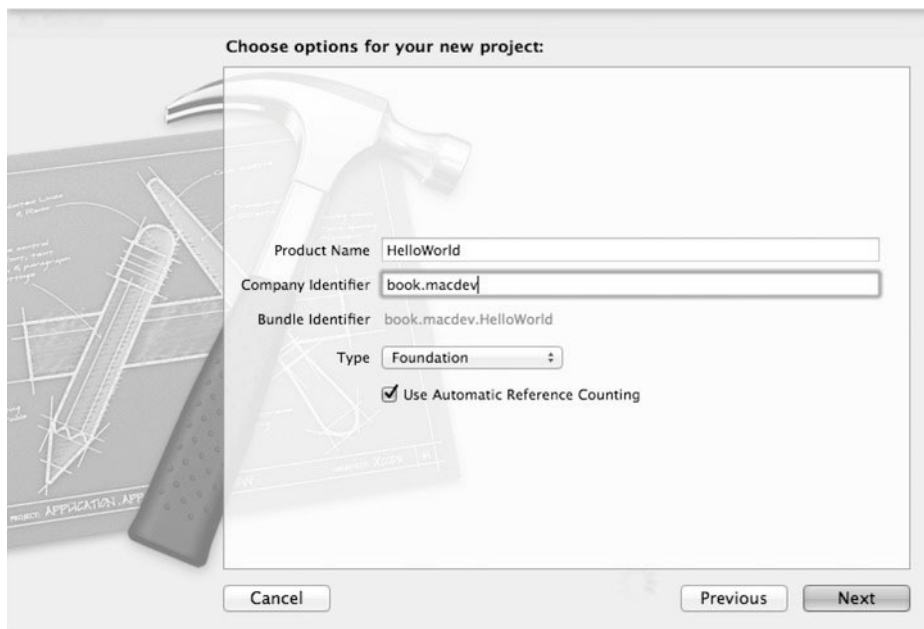


Figure 1-19. Naming your project and selecting its type

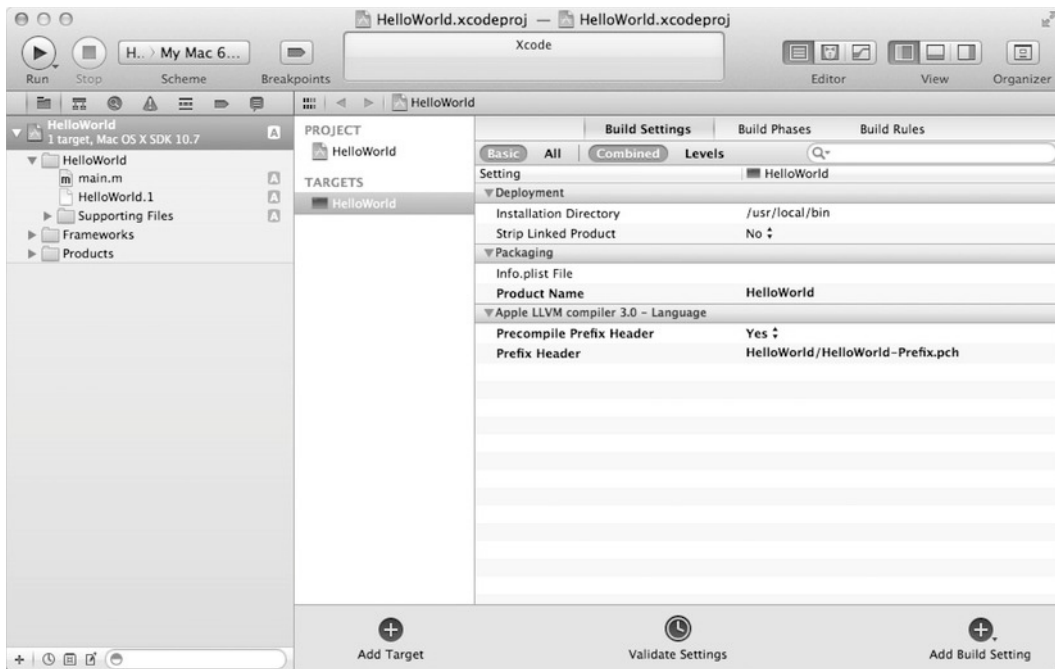


Figure 1–20. The Xcode window with the HelloWorld project

Expand all the folders on the left side of the screen and click the file `main.m` to view the generated source for this project. Also, click all three buttons above View in the toolbar across the top of the Xcode window. Refer to Figure 1–21 to understand the components of the Xcode window.

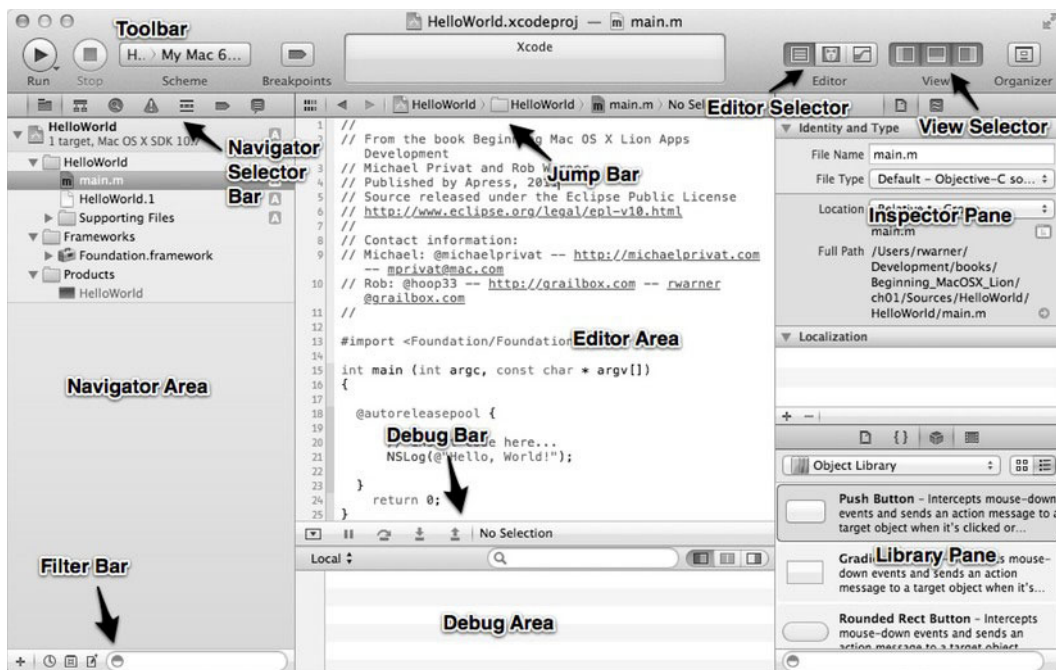


Figure 1–21. The components of the Xcode window

The busyness of the Xcode window can overwhelm developers, so we don't cover every aspect of every button, ribbon, or setting. Understanding the major areas of the Xcode window, however, can orient you sufficiently to give you an exploration context. Don't be afraid to click around and try things in Xcode to reveal more functionality than you realized was there. You can also hover over any of the various buttons in the Xcode window to display a tooltip that names and sometimes describes the control you're hovering over. The next few sections describe the major pieces of Xcode and prepare you for developing the Graphique project.

The Editor Area and the Jump Bar

The Editor Area sits in the middle of the Xcode window and usually dominates the focus of your interactions with Xcode. In the Editor Area, you edit the various files in your Xcode projects. This area changes the editor it uses to reflect the type of file you're editing. When you're editing a source code file, for example, it displays a modern source code editor with syntax highlighting, code completion, and the other features you'd expect from a programmer's text editor. If you're editing an Interface Builder file, the Interface Builder editor displays here. If you're editing a Core Data model, the Core Data Modeling tool takes over this area, and so on; the editor appropriate to the file you're editing displays.

The Jump Bar works with the Editor Area to allow you to jump to specific places in what you edit. You can, for example, jump to a specific file in your project and display it in the

Editor Area by clicking in the Jump Bar and selecting the file. You can also jump to a specific method in a file.

The Navigator Area and the Navigator Selector Bar

You navigate through your project in the Navigator Area. You use the Navigator Selector Bar to change which aspect of your project you want to navigate. When navigating through your project's files, for example, you select the Project navigator (the button on the upper left of the Navigator Selector Bar) to display the hierarchy of all the files related to your project. Clicking them opens them in the Editor Area.

You change the navigator that the Navigator Area displays by clicking the appropriate button in the Navigator Selector Bar. The navigators in Xcode are as follows:

- The Project navigator for navigating the files in your project
- The Symbol navigator for viewing your project's symbols such as classes and their methods
- The Search navigator for searching for specific text through all of your project's files
- The Issue navigator for displaying and jumping to the issues in your project (for example, compilation warnings or errors)
- The Debug navigator for displaying thread and other debug information when debugging your project
- The Breakpoint navigator for viewing all the breakpoints you've set in your project
- The Log navigator for viewing project logs, such as output from builds

The Filter Bar

Type text in the search field in the Filter Bar to dynamically filter what's displayed in the Navigation Area. If you're viewing the Project navigator, for example, for the HelloWorld project and you type **ma** in the search field, the Navigator Area hides everything that doesn't match the text *ma*, so only `main.m` displays.

The Filter Bar changes what it displays according to which navigator you've selected. When the Project navigator displays, for example, the Filter Bar shows a + button for adding a new file to the project.

The Debug Area and the Debug Bar

All Console output from your running code goes to the Debug Area. When debugging your project, you view the variables in your current debug context in the Variables View. The Variables View and the Console share the Debug Area, and you control what displays using the three buttons to the far right of the Debug Bar: one shows the

Variables View only, one shows both the Variables View and the Console, and one shows the Console only. The Debug Bar also gives you controls to use to step through your project when debugging.

The Inspector Pane, the Inspector Selector Bar, and the Library Pane

Together, the Inspector Pane and the Library Pane constitute the Utility Area. The Inspector Pane allows you to view and edit information specific to some item you've selected in the Editor Area. If you're editing a Core Data model, for example, you can view and edit information about the selected entity here. The Inspector Pane changes the buttons it displays across its top, depending on what kind of thing you're inspecting, so that you can switch among the various inspectors. This is an area where curious experimentation usually pays off.

The Library Pane shows libraries that you can interact with as you work on your projects. You can view libraries of code snippets, objects (useful when editing Interface Builder files), and others.

The View Selector

The View Selector allows you to show and hide the Navigator Area, Utility Area (the Inspector Pane and the Library Pane), and the Debug Area.

The Editor Selector

The Editor Selector has three buttons: one for showing the standard editor (for example, the source code editor for editing the currently selected file), the assistant editor (used to edit a file related to the one you're editing in the standard editor), and the version editor (used to compare different versions of your files as they change over time). Although you'll use the standard editor the most, you'll grow to rely on both the assistant and version editors for editing related files or understanding what has changed in files. Using the assistant editor, for example, you can display both an implementation file and its header file side by side.

The Toolbar

The Toolbar gives you something to click and drag to move your Xcode window around. It lets you choose what scheme you're currently using (for example, Debug vs. Release builds), lets you turn breakpoints on or off globally, shows you output about things like builds and issues, and gives you buttons to run, test, profile, analyze, and stop your application. Stop has its own button, while the other four operations share a button (click and hold the Run button to display a menu of the four buttons).

That covers the major areas of the Xcode interface. Click the Run button now to see your HelloWorld project build and run. You should see the text “Hello, World!” in the Console in the Debug Area, as shown in Figure 1–22.

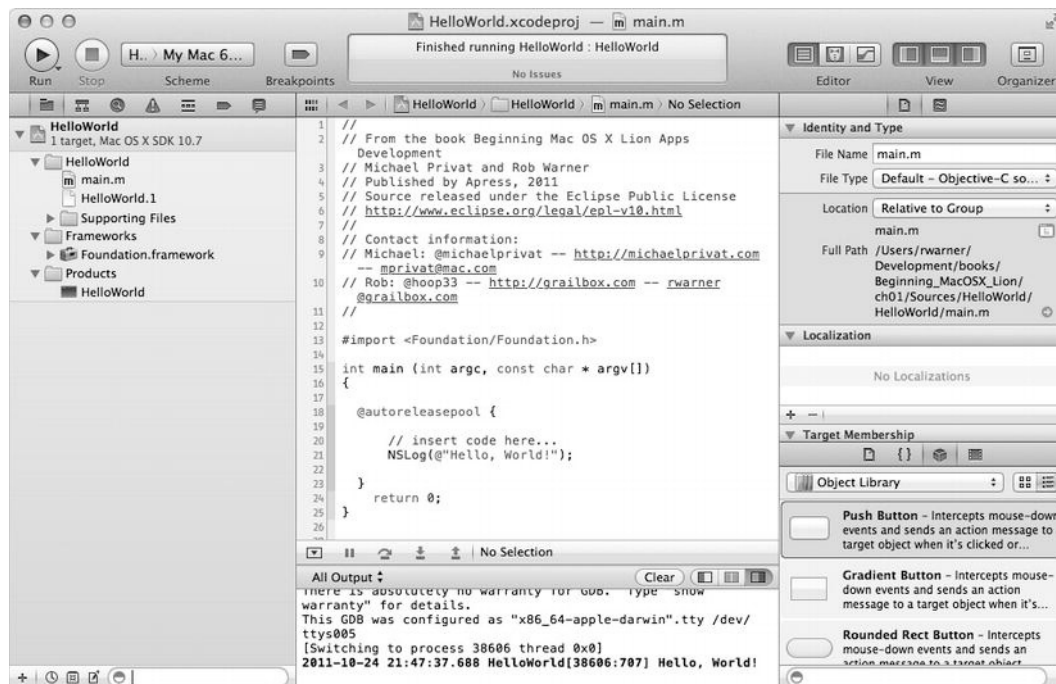


Figure 1–22. Output from the HelloWorld app

Creating a Project

Regardless of language and platform, creating the first piece of code of an application is often an exciting moment. It marks the point in time when the application morphs from concept to reality. In this section, we take this first step and create the first running shell of the Graphique application. In the vast majority of situations, the first few lines of code are the same regardless of the application. This isn't unique to Objective-C and Mac OS X. Every Java application, for instance, must have a public static void main(String[] args) method. Any Microsoft Windows application contains a fairly substantial amount of boilerplate code. Generally speaking, applications that run in a graphical environment require a larger amount of common initialization code. Thankfully, Xcode, like any respectable integrated development tool, generates all the setup for you. It does take away from the pleasure of typing the very first line of code, but as you create more and more applications, you will undoubtedly learn to appreciate the trade-off. Once Xcode is done generating the setup code, you will have a runnable application that will display a simple window. We then take you for a tour of what was created, why it was created, and where to go next.

To begin, launch Xcode and create a new project by selecting **File > New > New Project** from the menu. Note that you can also create a new project by pressing $\hat{+}+\mathbb{N}$. From the list of application templates, select the **Application** item under **Mac OS X** on the left, as depicted in Figure 1–23, and pick **Cocoa Application** on the right.

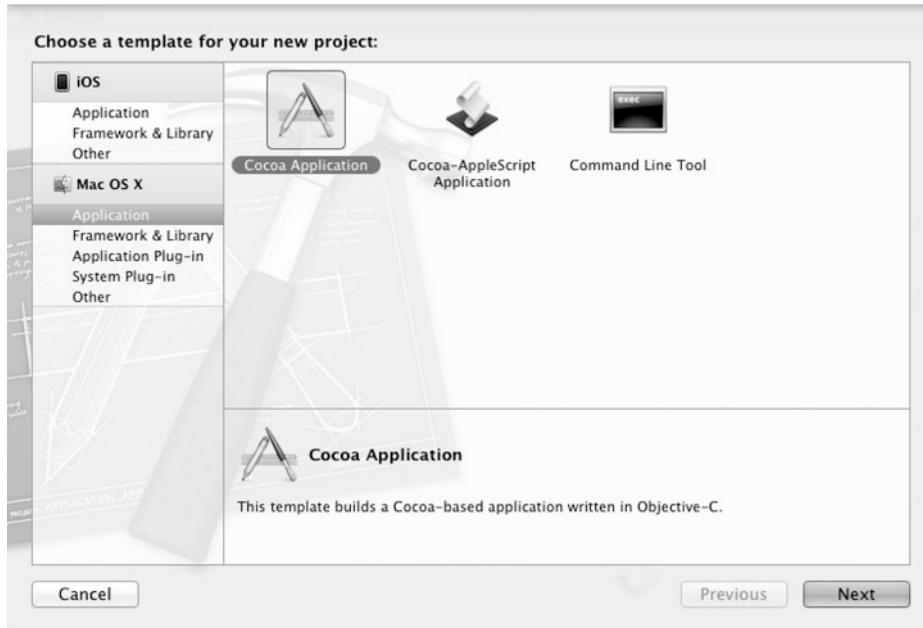


Figure 1–23. *Creating a new Cocoa application*

Click the **Next** button. The next screen presents you with options for configuring the code generator. For the Graphique app we build in this book, enter the following values:

- Product Name: Graphique
- Company Identifier: book.macdev
- Class Prefix: Graphique
- App Store Category: Utilities
- Create Document-Based Application: Unchecked (Document-based applications manage multiple documents and multiple windows, displaying a different document in each window. This is typically the case for applications such as word processors. Graphique, in contrast, is a single-document, single-window application.)
- Document Extension: Leave blank
- Use Core Data: Unchecked
- Use Automatic Reference Counting: Checked
- Include Unit Tests: Checked
- Include Spotlight Importer: Unchecked

Your screen should match Figure 1–24.

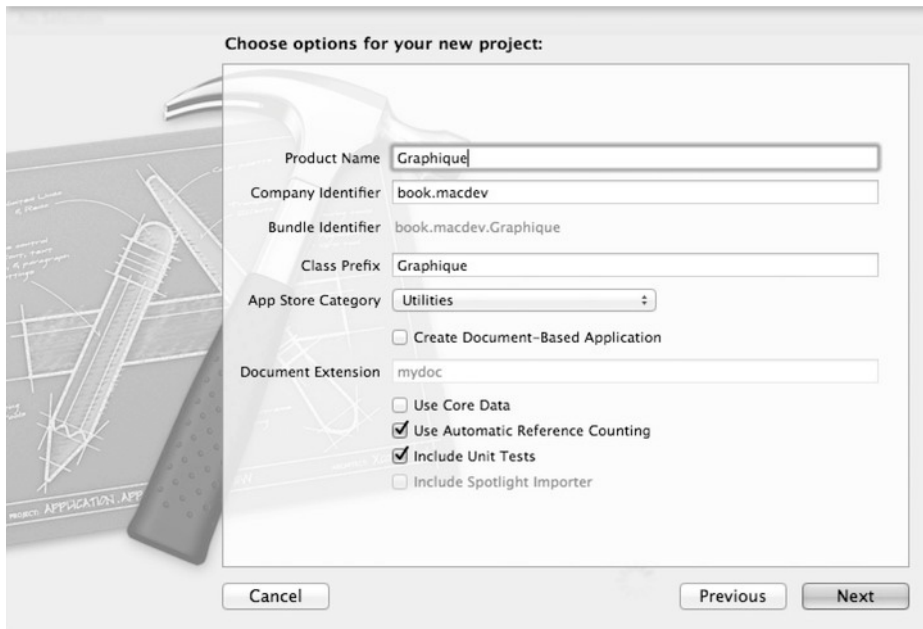


Figure 1–24. *The application options*

Clicking Next once the information is filled out pops up a save dialog asking for a location for where to put your project. Pick a folder and hit Save.

NOTE: The save dialog has a check mark that, if checked, will create a local Git repository. Although this is optional, we strongly recommend that you utilize a source-control repository when working on code. Not only will it help you keep your sanity when you try to track the change history of a file, but remote repositories also serve as excellent backups for when your workstation or laptop decides it has had enough with life.

Figure 1–25 shows the new project outfitted with all the necessary files to get started. In the next section, we explore the project and discuss what the artifacts generated are used for.

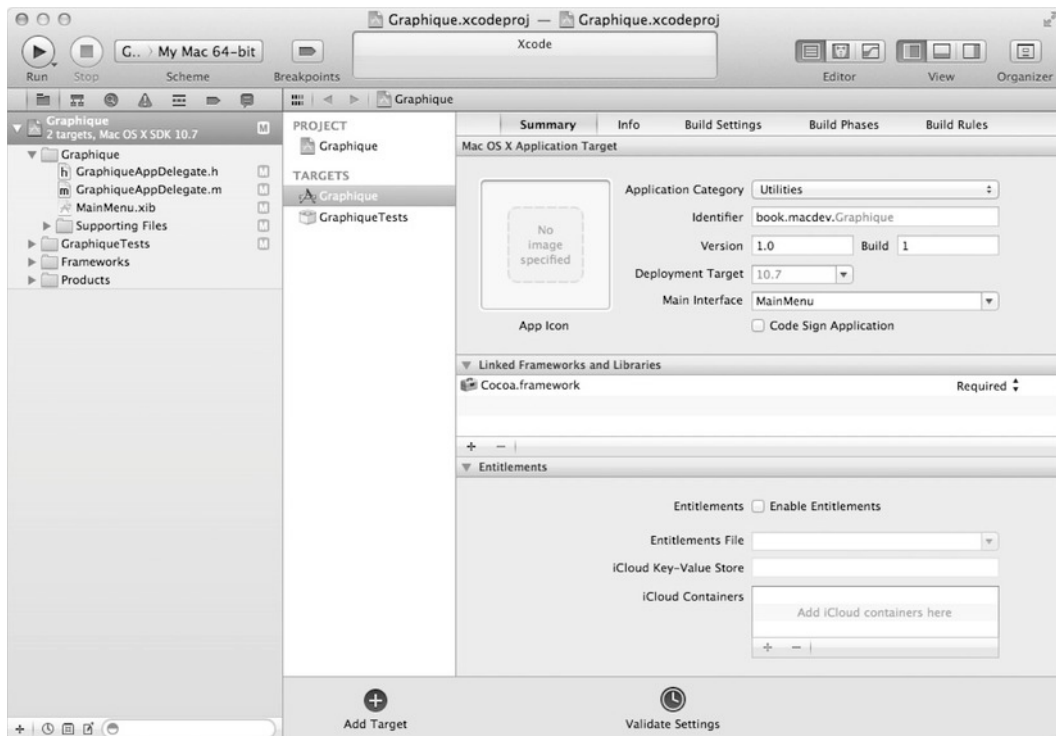


Figure 1–25. *The newly created Graphique application*

Before we start looking under the hood, however, let's indulge ourselves and see what the fruit of a few mouse clicks and key presses looks like. In Xcode, click the Run button to launch Graphique. The application launches, switches the menu bar to show the Graphique menu, and displays a blank window, as shown in Figure 1–26.

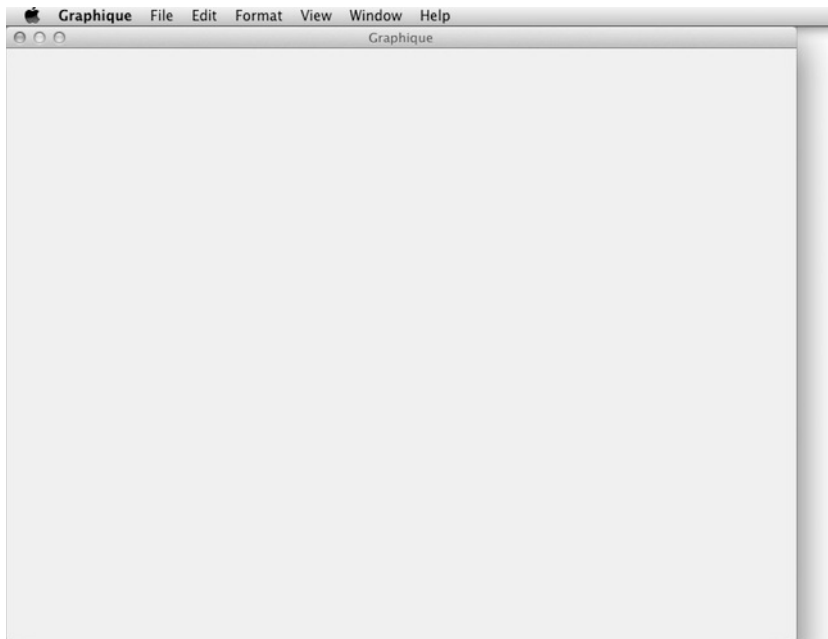


Figure 1–26. *The generated application running*

Although the application doesn't yet do anything useful, it offers some points of interest. The most obvious of these interesting points is that a window has opened so that the application is materialized on the screen. Another important detail is the existence of a menu in the system-wide menu bar. The menu has several default items; most of them are disabled, but some of them are active. If you select **Graphique > About Graphique**, you see the about box with some text in it. Also worth mentioning is that the **Graphique > Quit Graphique** menu item is working, including its keyboard shortcut ($\text{⌘}Q$).

It's time to quit drooling and to get to work. Use that **Graphique > Quit Graphique** option and return to Xcode.

Understanding the Major Components

Now that you've had a chance to run the application and see what you got from the Xcode application template, it's time to raise the hood and put our hands in the grease. Xcode generates a few files, and it can be overwhelming at first glance. The first thought that crosses the Xcode novice's mind is typically "Where do I go from here?" or "What's all that stuff?" In this section, we take a methodical approach to enumerating all the pieces so that we can get started making this application our own. Figure 1–27 shows what the Xcode artifacts should look like in the Graphique project.

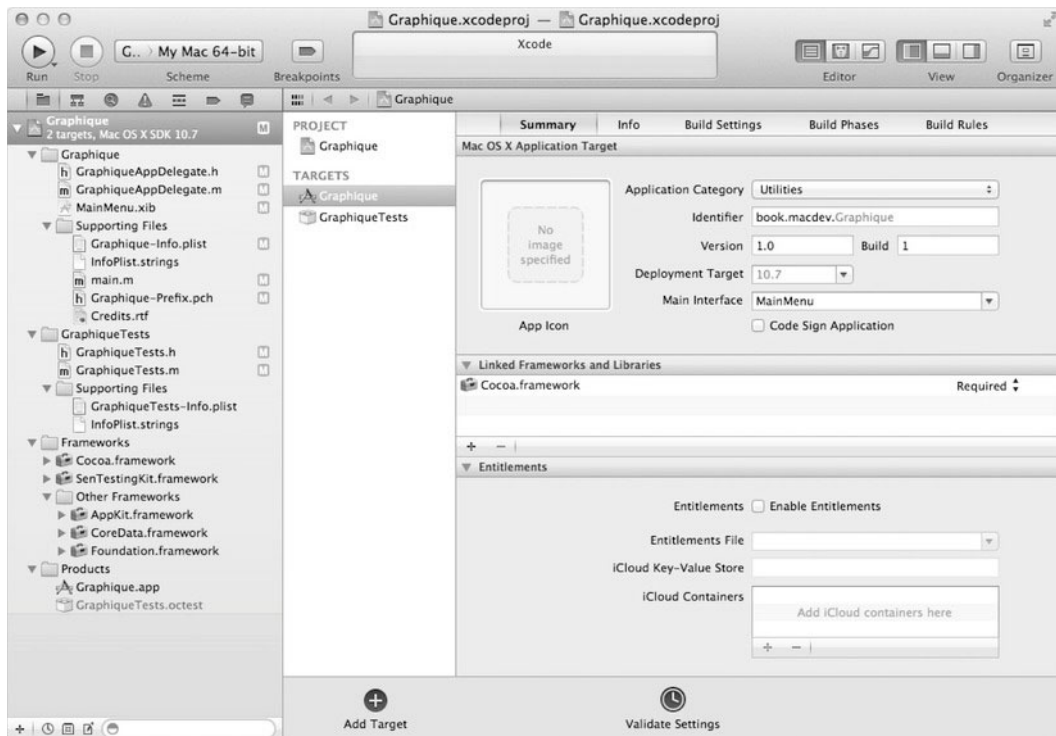


Figure 1-27. *The generated artifacts for a new Cocoa application*

The Project and Targets

In Xcode, artifacts are arranged in a tree structure. The top-level item—Graphique—represents the project. As you expect, every artifact belonging to the project is a subelement of Graphique in the tree structure, visually representing that the Graphique project contains all the other artifacts. A project defines two important sets of information:

- The set of files (source code and resources) available to the application build process
- The set of targets available to build the project into products

Targets define how to build your source files and resources into a packaged, runnable product. The runnable product can be the application itself but can also be a unit test suite. For now, we will set unit test suites aside and focus on the application product. This is the product that is built when you hit the Run button in Xcode.

Select Graphique in the Project navigator (in other words, the top level of the tree structure). In the editor window, you should see two targets listed: Graphique and GraphiqueTests. Select the Graphique target, as shown in Figure 1-27. The editor then shows more details about the selected target using several tabs. The default tab is the

Summary tab. It displays basic information about the product that will result from building this target. For the most part, it contains the information we entered in the project generation wizard in the previous section. The application icon is currently blank (which is materialized by a plain white icon at runtime). We won't worry about the icon for now and revisit it later in the book as we work through preparing the application for the Mac App Store.

Select the Build Phases tab. Build phases represent the steps in the build process. The default build phases are created for you, and in many cases, you won't need to add any new phase. The default phases list the source code files, frameworks, and resources that will be included in this build, as depicted in Figure 1–28.

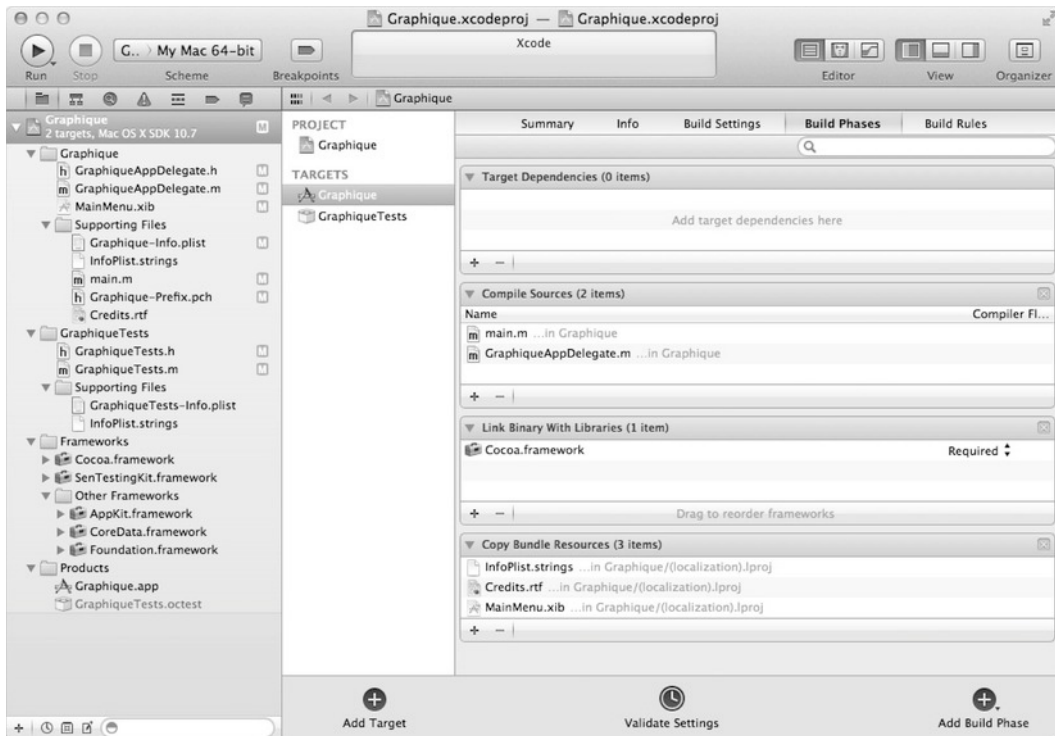


Figure 1–28. The Graphique target build phases

If you now select the GraphiqueTests phase and go to the Build Phases tab, you will notice, as shown in Figure 1–29, that the steps are slightly different. The first major difference is that the unit test target depends on the application target. This is to be expected since unit tests are all about testing the main application and therefore depend on the application building correctly. The last phase in this build target is called Run Script, which, as you might expect, runs an Xcode script that executes the unit tests.

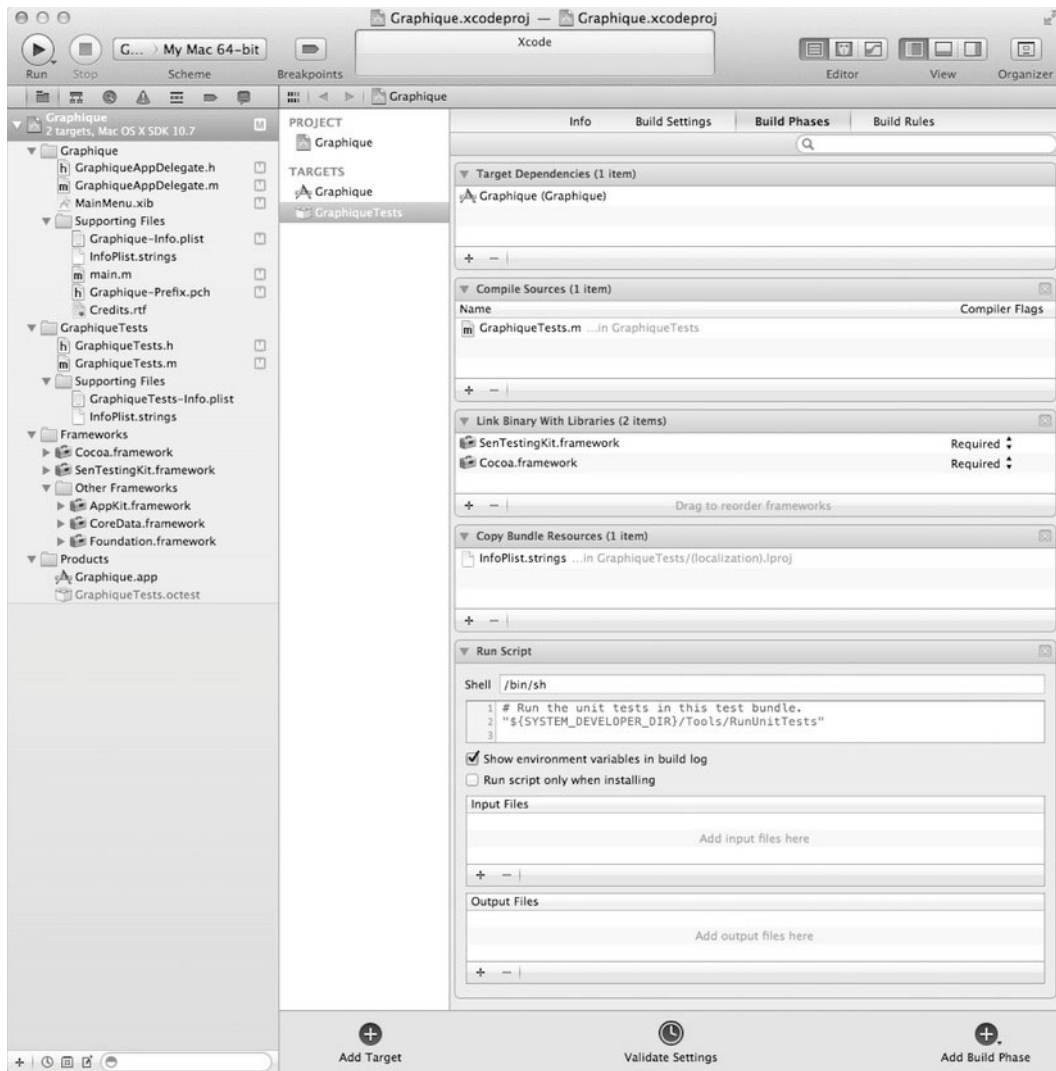


Figure 1–29. *The GraphiqueTests target build phases*

When the targets run, they produce the product, which then appears under the Products group in the Project navigator. The Graphique target generates a product called Graphique.app, which is the runnable application, while the GraphiqueTests target produces a product called GraphiqueTests.octest, which is the unit test suite.

The Application Architecture

In the remainder of this chapter, we will set the unit test target aside and focus on the application. Cocoa applications all have the same basic architecture. It is important to be familiar with it before getting started. Figure 1–30 illustrates this simple yet important structure.

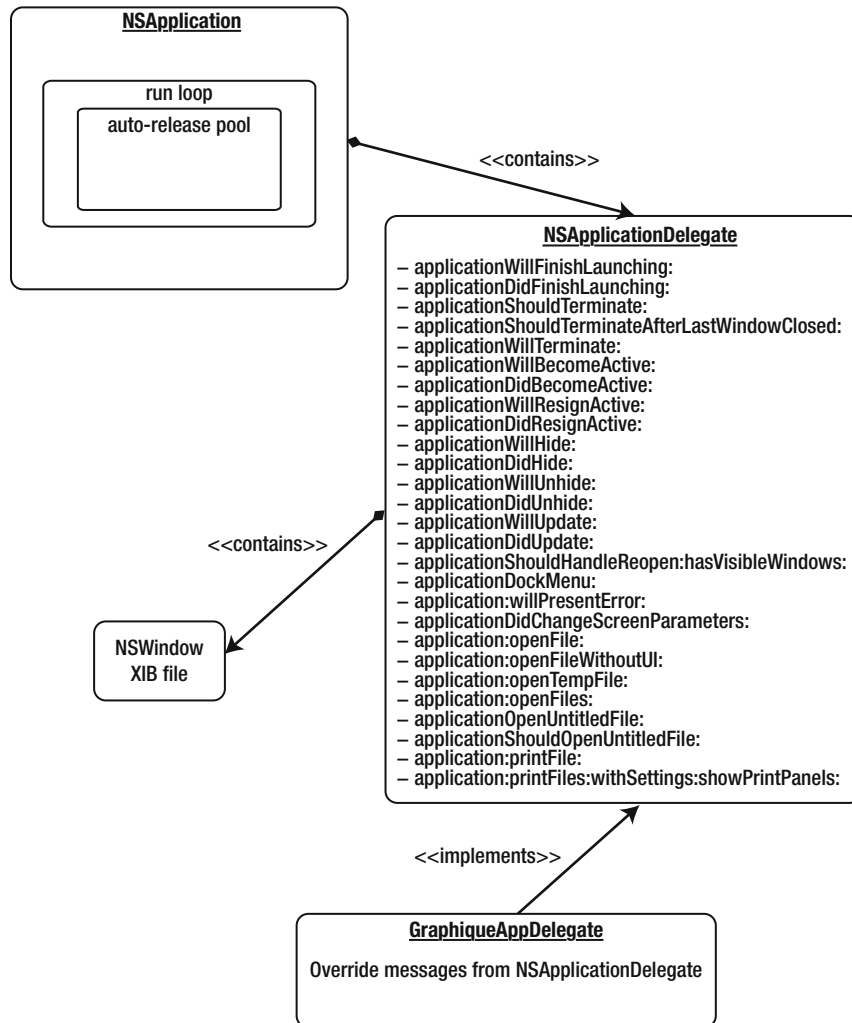


Figure 1–30. The basic application architecture

As a developer, you rarely find yourself having to extend the `NSApplication` class. Instead, you implement the `NSApplicationDelegate` protocol and let the `NSApplication` class manage the run loop and the rest of the application environment, such as the autorelease pool that handles garbage collection. As the application goes through its life

cycle (start, stop, minimize, hide, and so on), the `NSApplicationDelegate` is notified, and your code can react to these notifications.

The application delegate also declares the window attribute, which is set through the graphical interface builder and defines the `NSWindow` instance to use to materialize the window. We explore the user interface objects in depth in the next chapters. For now, just remember that the application delegate defines the window for the application.

The Source Code and Resources

All the source code and resources used in the application are organized under the `Graphique` group, as shown in Figure 1–31.

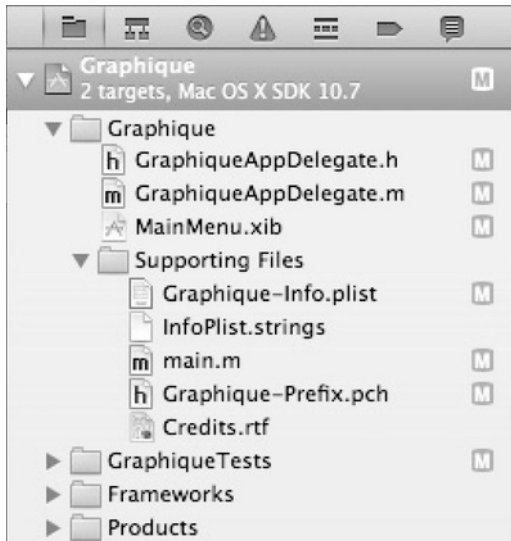


Figure 1–31. *The application source code and resources*

Working your way through the artifacts and understanding how they all fit together can be a maze. But just like Theseus walked out of the Labyrinth by following the string he had attached to the entrance and unwound as he advanced, we start from the place we are now and follow the linkages and dependencies.

Select the `Graphique` project in the navigator and go to the Summary tab of the `Graphique` target. There it shows, in the Main Interface drop-down, that an artifact called `MainMenu` drives the main interface. This is the starting point of our exploration. Go back to the Project navigator, find `MainMenu.xib`, and select it.

NOTE: XIB files, often called *nibs* because they are an evolution of the NIB files that had a .nib extension, contain the user interface layout information. Prior to Xcode 4, XIB files were opened in a separate application called Interface Builder (IB). As of Xcode 4, IB has been integrated into Xcode.

Although you can also lay out user interfaces programmatically, Interface Builder is a convenient point-and-click way to build your user interfaces. In the next chapter, we explain how to use Interface Builder. Figure 1–32 illustrates what you should see when you select `MainMenu.xib`. Make sure you select the Utilities button at the top-right corner of the Xcode IDE to see the right sidebar (in other words, the Utilities panel).

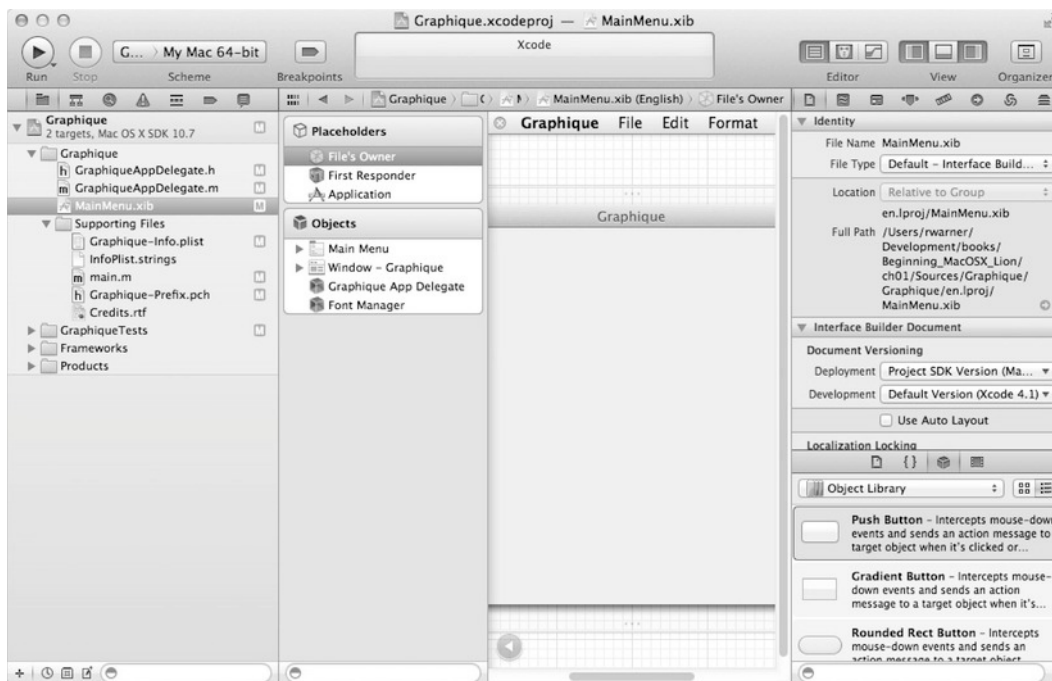


Figure 1–32. The `MainMenu.xib` file in Interface Builder

Interface Builder offers two types of entities:

- Placeholders represent objects that exist within the runtime environment.
- Objects represent objects that will be created when the XIB file is run.

If you select the File's Owner placeholder and select the Identity inspector tab in the Utilities panel, you will notice that the Interface Builder file is owned by an object of type `NSApplication`, which, as you may expect, is the root class for all applications. A Mac OS X application does not extend `NSApplication` directly. Instead, it is driven by setting the delegate property of `NSApplication` to a class that implements the

UIApplicationDelegate protocol. The `NSApplication` class makes the appropriate calls to its delegate throughout its life cycle (startup, shutdown, and so on). This is where the application-specific code begins. With the File's Owner (the `NSApplication` instance) still selected, go to the Connections inspector to see what the delegate is linked to. Figure 1–33 shows the Connections inspector tab selected.

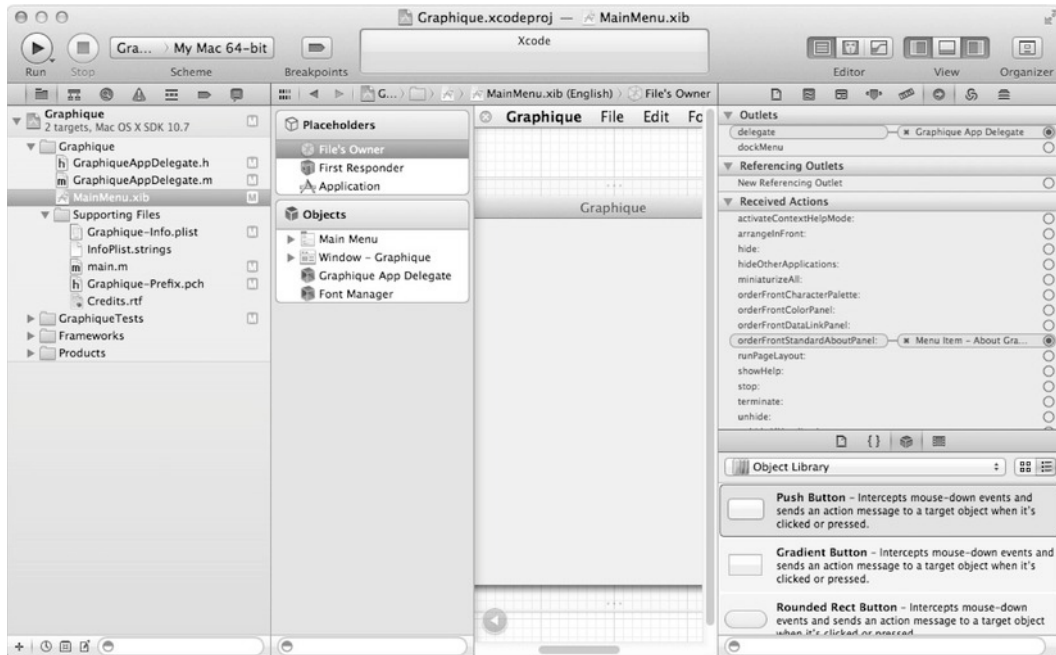


Figure 1–33. The Connections inspector showing the application delegate link

The connection shows the delegate set to the object named `Graphique App Delegate`. Select this object in the Objects section and go to the Identity inspector, as shown in Figure 1–34. Notice that this object is materialized in the application as a `GraphiqueAppDelegate` class.

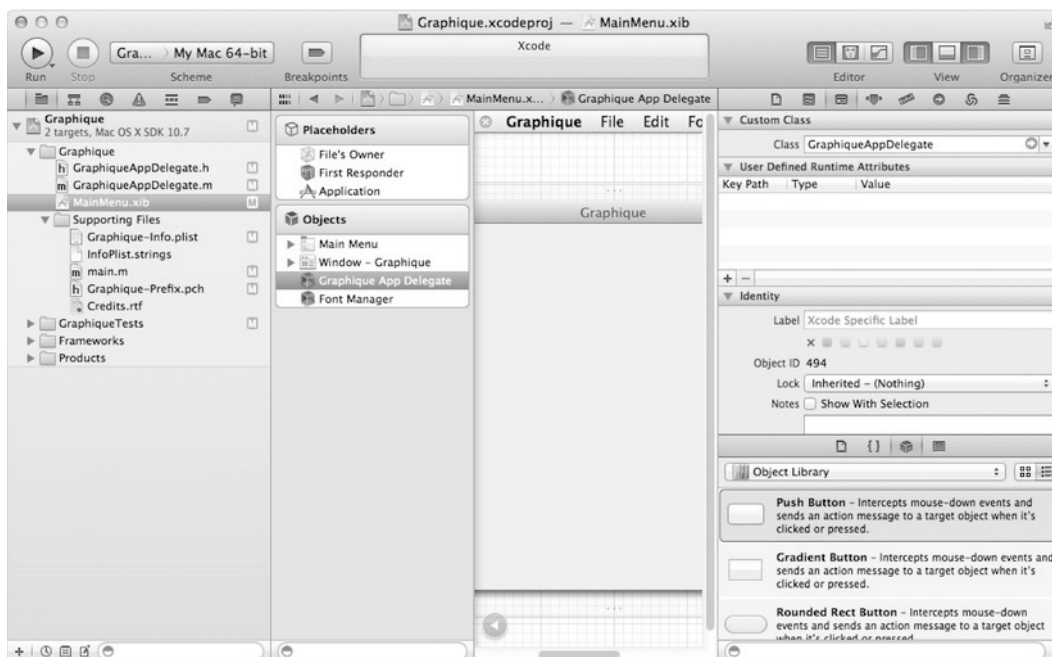


Figure 1–34. Identifying the class used for an IB object

We're almost at the beginning of the thread we're following. Open `GraphiqueAppDelegate.m` and find the `applicationDidFinishLaunching:` method. Xcode has conveniently inserted a comment stating "Insert code here to initialize your application." This is where the thread we're following starts. Go ahead and add a log statement to validate that we found the right method. Since no technical book is complete without a couple of references to the enduring "Hello World!" statement, edit the `applicationDidFinishLaunching:` method to look like this:

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{
    NSLog(@"Hello World!");
}
```

Launch the application. The console automatically appears because you logged something to it, and it displays a log message similar to the one shown here:

```
2011-08-26 06:26:33.685 Graphique[2960:407] Hello World!
```

Take a closer look at the `NSApplicationDelegate` protocol. It contains other methods called at different points in the application's life cycle. `NSApplicationDelegate` methods are called by the default notification center as the application interacts with the Mac OS X environment.

NOTE: You can always access the documentation quickly from the source code editor by using the Option+click shortcut. Open GraphiqueAppDelegate.h and Option+click the UIApplicationDelegate protocol to open the quick help view, as depicted in Figure 1–35. To get more in-depth documentation, click the book icon on the top right of the quick help window.

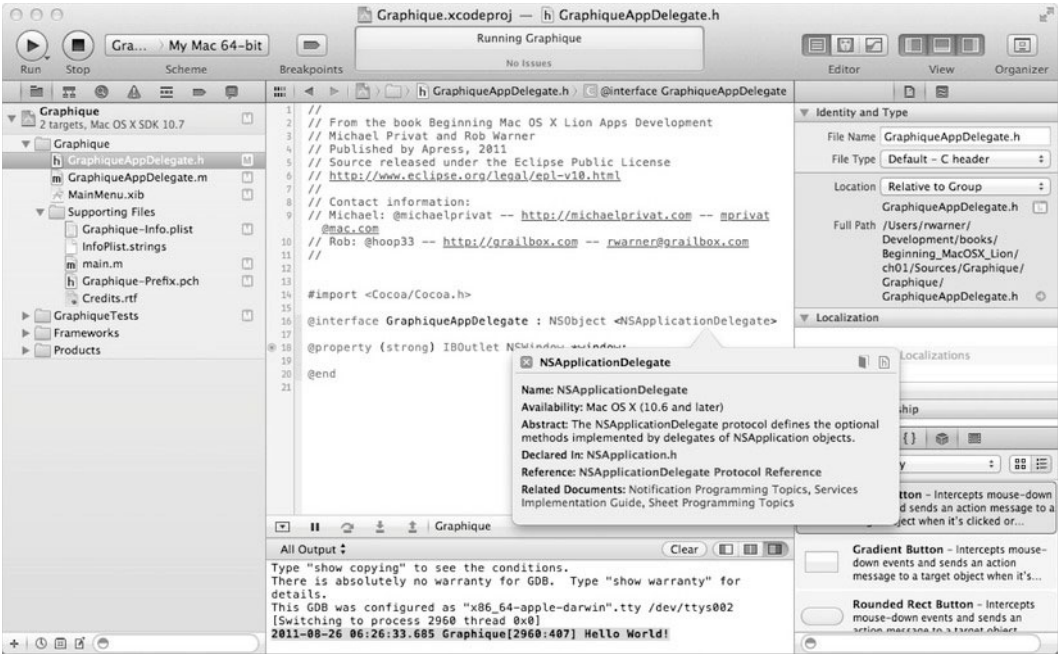


Figure 1–35. Accessing the quick documentation from the source code editor

You can be notified when your application becomes active, when it is about to quit, and so on. Override these methods in GraphiqueAppDelegate.m to have Graphique’s underlying UIApplication instance invoke your code. Table 1–1 shows the available life cycle messages that your delegate can receive.

Table 1–1. The UIApplicationDelegate Protocol Messages Dealing with the Application Life Cycle

Method	Description
- (void)applicationWillFinishLaunching:(NSNotification*)aNotification	This is sent immediately before the application object is initialized.
- (void)applicationDidFinishLaunching:(NSNotification*)aNotification	This is sent immediately after the application object is initialized. This is the method you typically override to place your own initialization code.

Method	Description
- (NSApplicationTerminateReply) applicationShouldTerminate:(NSApplication*)sender	This is sent to the application when the operating system wants to close it or because the <code>terminate:</code> method has been invoked. You typically return <code>NSTerminateNow</code> to let the application close or <code>NSTerminateCancel</code> if you want to delay the closing, for example if you have cleanup work to do.
- (BOOL) applicationShouldTerminateAfterLastWindowClosed:(NSApplication *)theApplication	This method is called when the last window of the application has been closed. If you want your application to terminate, override this method in your delegate class and return <code>YES</code> , else return <code>NO</code> . By default, it returns <code>NO</code> .
- (void) applicationWillTerminate:(NSNotification*)aNotification	Called just before terminating the application. At this point, you no longer have a say on whether the application will terminate, but you still have an opportunity to do some work.
- (void) applicationWillBecomeActive:(NSNotification*)aNotification	Sent immediately before the application becomes active. An application becomes active when it is brought in front of all other running applications.
- (void) applicationDidBecomeActive:(NSNotification*)aNotification	Sent immediately after the application becomes active.
- (void) applicationWillResignActive:(NSNotification*)aNotification	Sent immediately before the application loses its active status.
- (void) applicationDidResignActive:(NSNotification*)aNotification	Sent immediately after the application loses its active status.

Plenty of other delegate methods are available. Although we touch on a few of them later in this book, we encourage you to look at the `NSApplicationDelegate` documentation to see them all.

Let's play around a little bit more and intercept one of these messages to clearly illustrate how this works. Open `GraphiqueAppDelegate.m` and add the methods shown in Listing 1-1.

Listing 1–1. *NSApplicationDelegate method implementations*

```
- (void)applicationDidBecomeActive:(NSNotification *)aNotification {
    NSLog(@"Application is active");
}

- (void)applicationDidResignActive:(NSNotification *)aNotification {
    NSLog(@"Application is no longer active");
}
```

In effect, this is telling the Mac OS X environment that you want to intercept these messages. Start the Graphique application. As the window appears, you get the following log:

```
2011-08-26 06:34:14.479 Graphique[3077:407] Hello World!
2011-08-26 06:34:14.647 Graphique[3077:407] Application is active
```

Now switch to another application, and the following line is appended to the log:

```
2011-08-26 06:34:16.938 Graphique[3077:407] Application is no longer active
```

The About Dialog

One last piece of magic provided by the Xcode generator is the about dialog. While Graphique is running, select **Graphique > About Graphique** in the menu to see the about dialog. This is the default implementation provided by the `orderFrontStandardAboutPanel:` method in the `NSApplication` class. To track down how the About Graphique menu item is linked to that method, open `MainMenu.xib` in Interface Builder. Remember that the File's Owner placeholder is actually the `NSApplication` instance. Expand the Main Menu object to see Menu Item - About Graphique, as shown in Figure 1–36. In the Connections inspector, you can see that this menu item is sending a message to the File's Owner (`NSApplication`) `orderFrontStandardAboutPanel:` method.



Figure 1–36. *The about box menu item*

The implementation of `orderFrontStandardAboutPanel:` creates a simple dialog box and uses the content of the `Credits.rtf` file that is present in the project. Find that file, open it, and see that it matches the content of the about dialog.

Summary

In this chapter, you got your hands on Xcode 4, installed it, and learned how to use it. You learned about the major components of the Xcode interface, navigated through them, and saw how to build and launch an application. You also started building the Graphique app and learned about some of the libraries, including Cocoa, that you'll use when building Mac OS X applications.

Although your Graphique application builds and runs, you've done precious little software development. Instead, you've been able to rely on Xcode's code generation and templates to get a working application. In the next chapter, however, you'll dive in to using Xcode's editors, including Interface Builder, to lay out the Graphique user interface and bind the user interface to code.

Laying Out the User Interface

The appearance of an application often conveys an indelible first impression on users' minds, whether positive or negative, that strongly influences whether the application becomes a hit or simply moves to the virtual trash. Providing an attractive and usable interface remains one of the hallmarks of top applications.

Mac OS X offers an array of user interface widgets that, when intelligently stitched together, fuse to provide stunning and useful interfaces. In this chapter, we explore some of these widgets and how to incorporate them into the Graphique application. The widgets we explore include the following:

- Labels, which display static text
- Text fields, which allow users to enter text
- Buttons, used to let users initiate some action
- Toolbars, which display a strip of buttons with images
- Split views, which divide a view into two separate views
- Sliders, which allow users to select a value by dragging a “thumb” across a range of values
- Tables, which display tabular data

This chapter focuses only on using Mac OS X's widgets to create the user interface, not on responding to user input or reacting to events. Chapter 3 will tackle those topics.

Through this chapter, you build an interface that displays three different views, separated by split views:

- A view to enter equations
- A table of x , y values for an equation
- A tree that shows recently used equations

At the end of the chapter, the Graphique application will look like Figure 2–1.

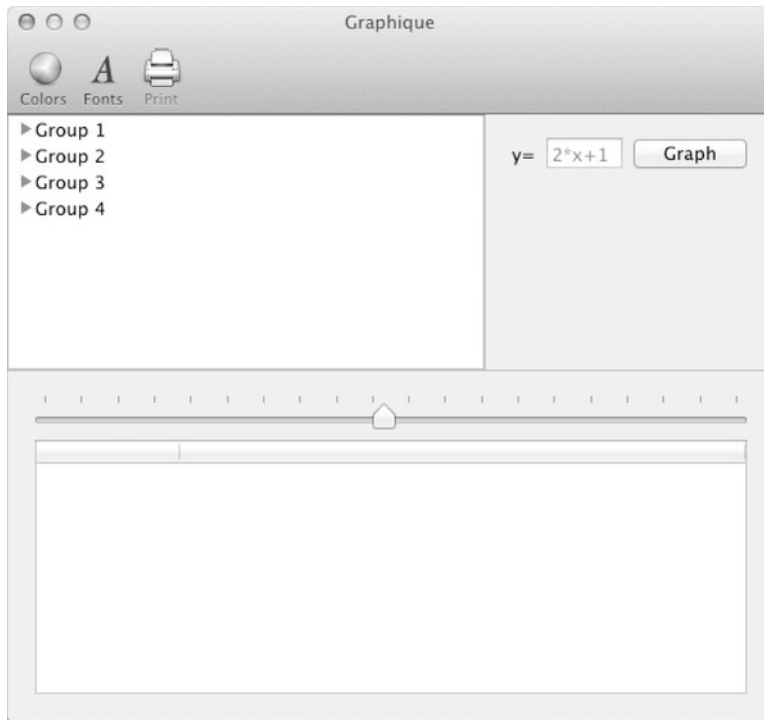


Figure 2–1. *The Graphique application when the layout is complete*

Creating the Split View

In the Graphique application, we want to give users a place to enter an equation. We also want to show users the graph that an equation generates. Finally, we want to show users a list of the equations they've already entered so that they can quickly retrieve equations and their resulting graphs. The user interface has one window, but we've outlined three different views that we want that window to display. Cocoa offers a class called `NSSplitView` that allows you to show two different views in the same window, with a movable separator between them. The separator can run horizontally or vertically, and you can also nest them to create multiple views in a window, all with movable separators between. We'll use the `NSSplitView` class to address the user interface needs of Graphique.

We refer to the three views that we want to display in the Graphique application like this:

- *Equation Entry View*: The place users can enter an equation
- *Recent Equations View*: The list of equations users have already entered
- *Graph View*: The graph represented by the equation in the Equation Entry View

We arrange these three views as shown in Figure 2–2, with a vertical separator, or splitter, between the Equation Entry View and the Recent Equations View, and a horizontal separator (splitter) between the top two views and the Graph View.

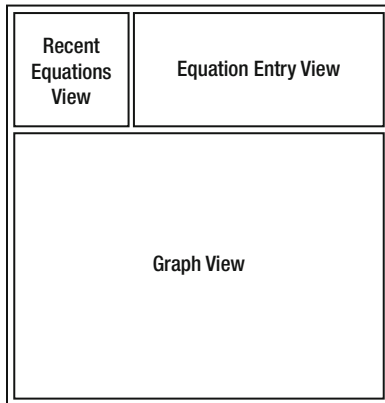


Figure 2–2. *The layout of the Graphique user interface*

Creating the Horizontal NSSplitView

To begin creating the desired view layout for Graphique, we first divide the window horizontally, with a view above and a view below and a movable separator between. Open `MainMenu.xib` in Xcode and select the Window – Graphique object from the Object hierarchy. You may need to click the button to show the Document Outline, as shown in Figure 2–3, to get your view to match ours. The Document Outline helps you understand the relationships among the various views.

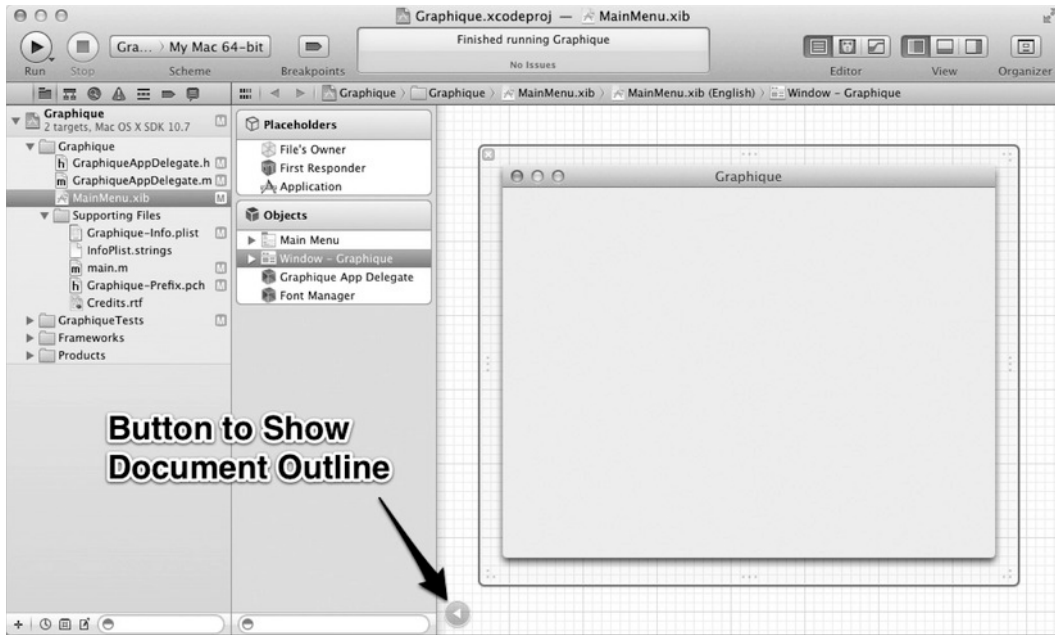


Figure 2-3. The main Graphique window and the button to show the Document Outline

Expand the Window object in the Object hierarchy to see the View object it contains, and then select the View object, as shown in Figure 2-4.

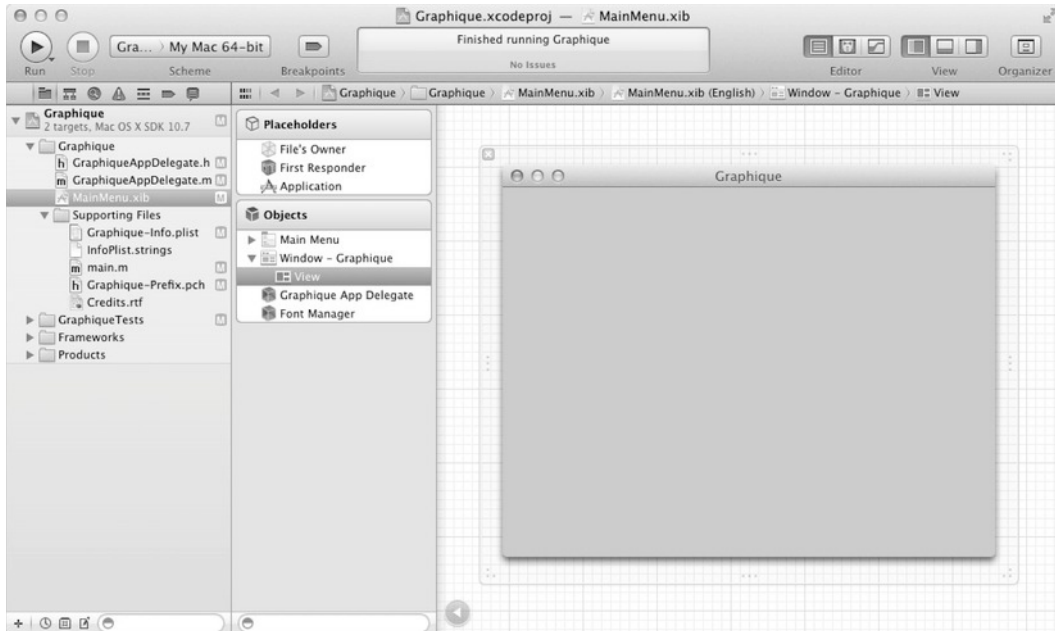


Figure 2-4. The main Graphique window with the view selected

Show the Utilities panel (on the right side of Xcode), show the Object Library in that panel, and type **horizontal** in the search field to show the Horizontal Split View object in the library, as Figure 2–5 shows. Drag a Horizontal Split View object onto the Graphique window and drop it. It should appear both in the Graphique window and in the Object view, below the view that's below the Graphique window, as shown in Figure 2–6. You should also see the two custom views that the split view contains.



Figure 2–5. *The Horizontal Split View object in the Object Library*



Figure 2–6. *The Horizontal Split View object added to the view in the Graphique window*

Now, resize the Horizontal Split View object to completely fill the window. We also want to make sure it continues to fill the window, even if the user resizes the window, so show the Size inspector and select all the springs and all the struts in the Autosizing view. In other words, click the two double-headed arrows inside the square and the four bars outside the square so they're all solid, as shown in Figure 2-7.

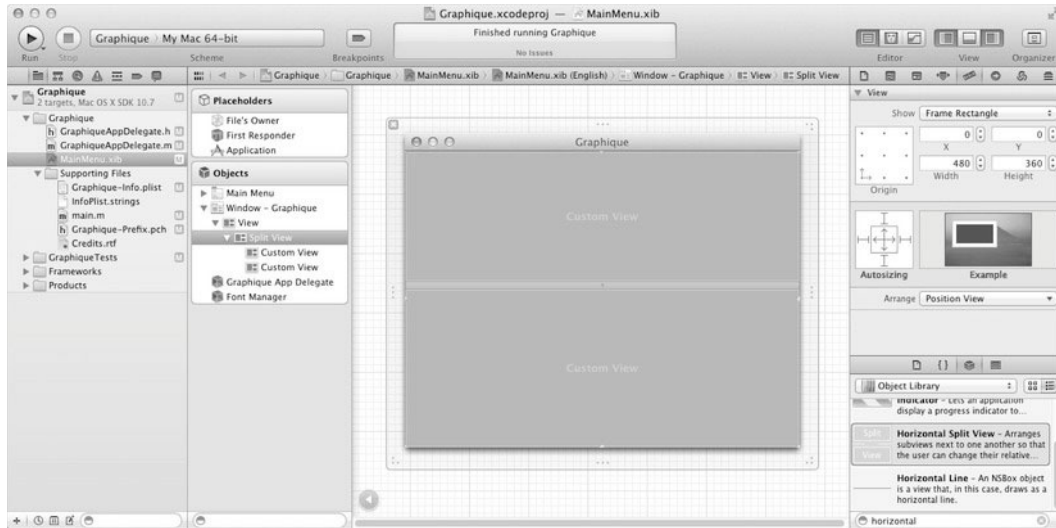


Figure 2-7. Setting the Horizontal Split View object to size with the window

Run the application. You should see a window split in half horizontally, as shown in Figure 2-8. Resize the window and make sure the split view continues to fill the view.



Figure 2–8. *The Graphique application with a Horizontal Split View object*

Creating the Vertical NSSplitView

The next step is to split the upper half of the Graphique window with a vertical split view. Go back to Xcode and drag a Vertical Split View object from the Object Library onto the upper half of the horizontal split view you just created. Drag and size it to fill the entire upper half of the horizontal split view, and select all its springs and struts in the Autosizing box so that it all looks like Figure 2–9.

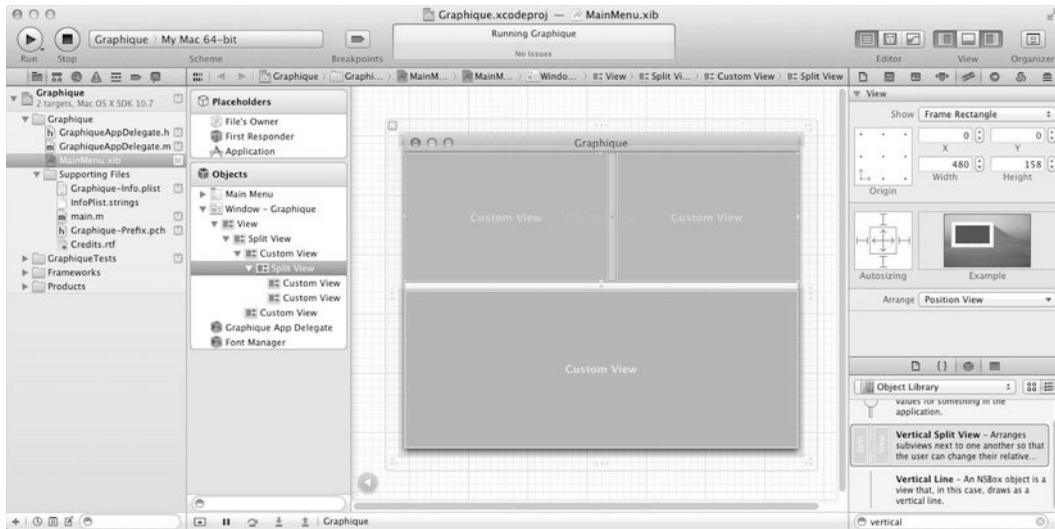


Figure 2-9. Adding a Vertical Split View object

Again, build and run the application. You should see a window that looks like Figure 2-10. Resize the window to make sure the split views continue to fill the window, and move the splitters around to confirm that they move.

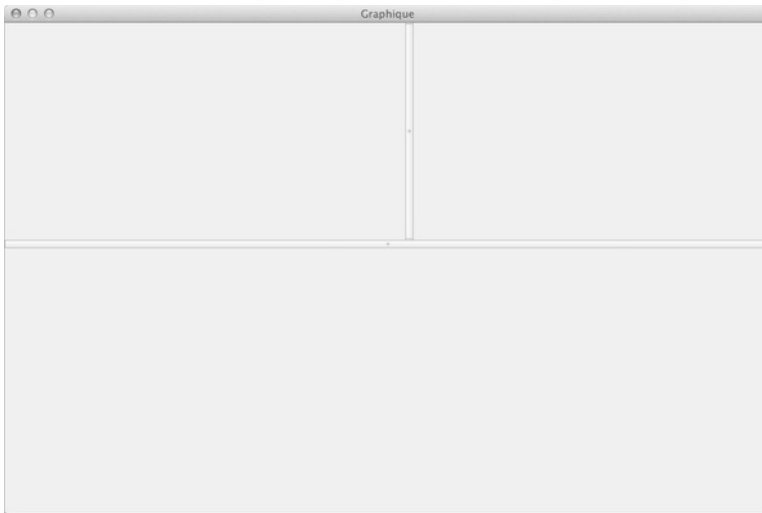


Figure 2-10. The Graphique application with both Horizontal and Vertical Split View objects

We want to take one more step before moving on. Those splitters seem so thick and heavy, and the trend in Mac OS X applications points to using thin splitters. Go back to Xcode and select the Horizontal Split View object in the object hierarchy, go to the Attributes inspector, and change the style from Pane splitter to Thin divider. Then, do the same for your Vertical Split View object. When you build and run the application, it should resemble Figure 2-11.

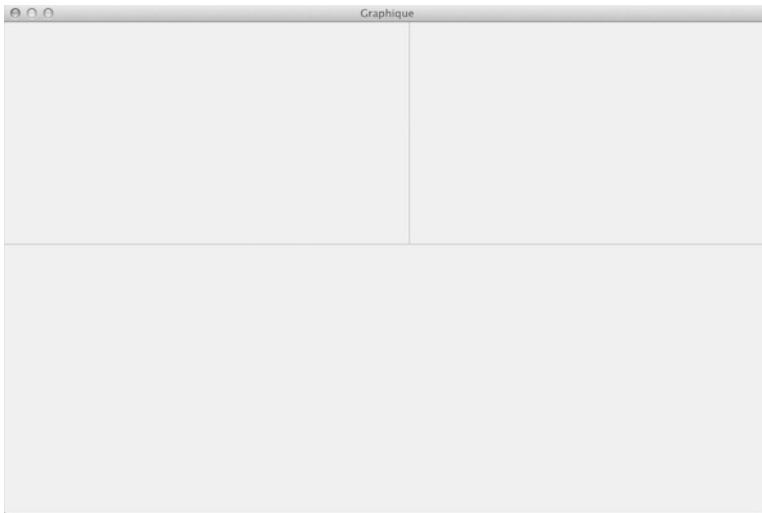


Figure 2–11. *The Graphique application with thin dividers*

You’ve created the structure of the Graphique application layout. Next, you must create the actual views that the splitters will divide and swap them for the placeholder views. Read on to the next section to understand how to do that.

Creating the Equation Entry Panel

So far, you’ve learned how to do some basic layout. The real power of Xcode comes from the ability to break down the user interface into components. In this section, we build our first version of the equation entry component. The purpose of this component is to allow the user to specify an equation to graph. There are two important reasons for creating a component instead of just adding more individual widgets to the MainMenu.xib file:

- *Code reuse:* Creating a clearly separated component allows you to reuse it in various contexts in your application. Think about the Cover Flow user interface component on your Mac. It is used in iTunes, but the same component is also used in Finder.
- *Code isolation:* All the code pertaining to your component’s internal function is kept within the component and not in some generic controller class. This allows you to improve the component in the future without impacting the rest of the application. In Chapter 4, we put this in practice by enhancing the equation entry component.

For this first version, we keep the component trivial. We’ll have a label with the text “y=” and a simple text entry field where the user will type the equation as a function of x.

Using NSViewController

If you come from an iOS development background, you are already familiar with the all-important `UIViewController` class. The `NSViewController` class was introduced with Mac OS 10.5 (Leopard). Although it sounds like a parallel with iOS's `UIViewController`, it is not nearly as central. It is meant to manage custom views but not necessarily receive events from them, as is customary in the iOS world. The primary function of the `NSViewController` class is to materialize custom views from XIB files so that you can use them in your code. The `NSViewController` controller has a `view` attribute that it populates from the XIB file automatically.

Creating a new custom user interface component consists of creating a new `NSViewController` subclass and laying out its view in Interface Builder.

To keep things organized, create a group to store all the custom components by going to Xcode and doing the following:

1. Highlight the Graphique folder, and select **File ► New ► New Group** from the menu.
2. Call the new group **Views**.
3. Select the new group and create a new class by selecting **File ► New ► New File...** from the menu.
4. From the Mac OS X/Cocoa section, select Objective-C class, as illustrated in Figure 2–12, and click Next.
5. On the following screen, enter `EquationEntryViewController` for Class and select or type `NSViewController` as the superclass so it matches Figure 2–13. Click Next.
6. Make sure the Views group and the Graphique target are selected, and click Create.

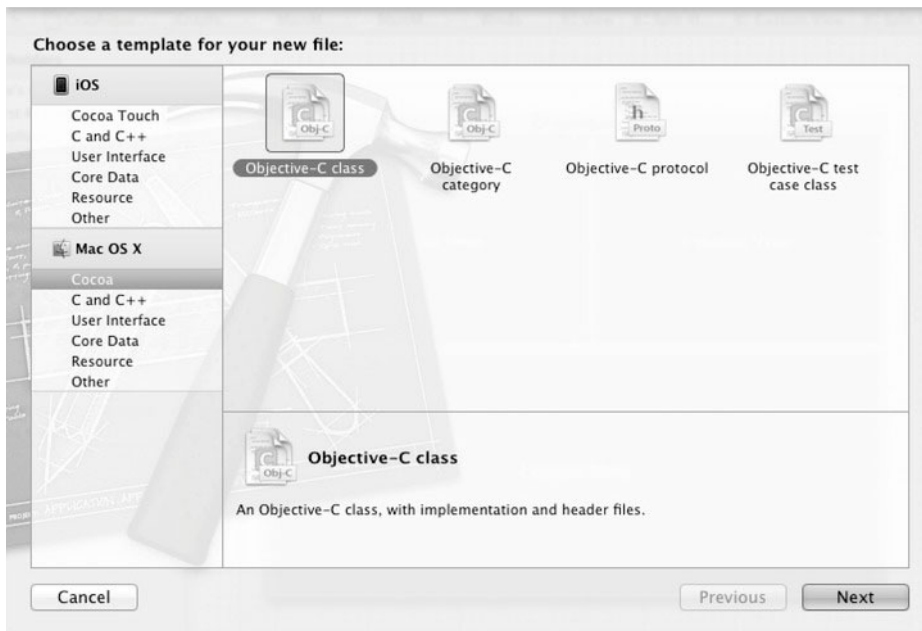


Figure 2-12. Creating a new Objective-C class for the Equation Entry View controller

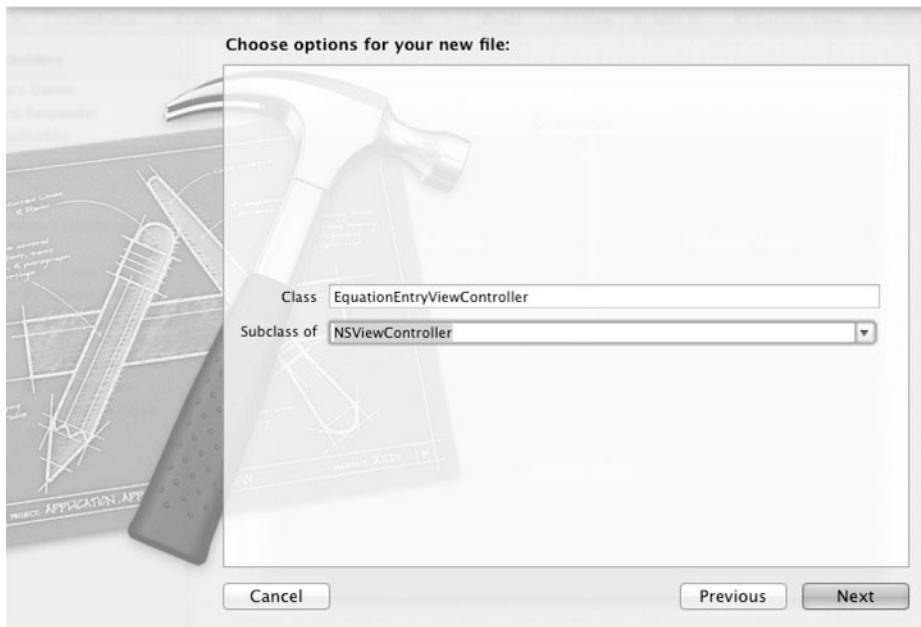


Figure 2-13. Making the new view a subclass of NSViewController

This process should have generated three files: `EquationEntryViewController.h`, `EquationEntryViewController.m`, and `EquationEntryViewController.xib`, as illustrated in Figure 2–14.

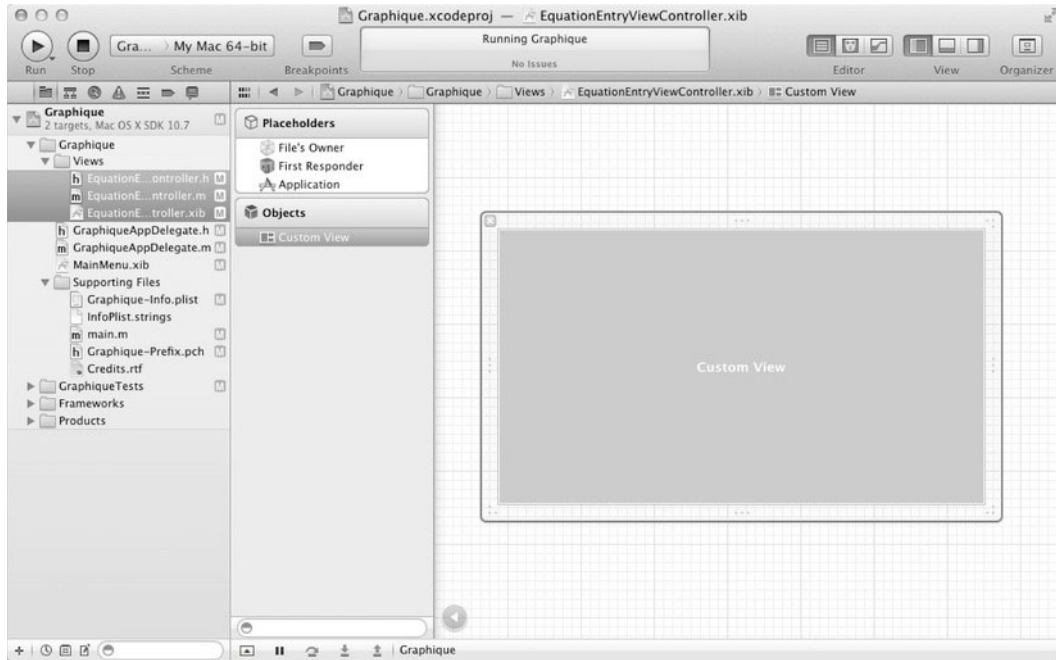


Figure 2–14. The generated files for the new custom component

Laying Out the Custom Equation Entry Component

Select `EquationEntryViewController.xib` to begin laying out its view. You can start dragging user interface components from the library onto it just like you did in the previous section. The only difference here is that you have isolated your custom component so you need to worry only about it and nothing else. This is code isolation.

If you select the File's Owner and look in the Identity inspector, you can see that the owner is the `EquationEntryViewController` class. Now select the Connections inspector tab and notice how the controller's view outlet is linked to the Custom View object. This is how the controller knows how to display its view.

Now let's finish laying things out. Add a label, a text field, and a push button. Change the label text to **y=** and the title on the push button to **Graph**. Size the view to fit, as shown in Figure 2–15. The user will be expected to type in a function of *x* and hit the Graph button in order for it to do anything useful. Handling events like button presses will be the focus of the next chapter. For now, we're simply planning and laying out our user interface.

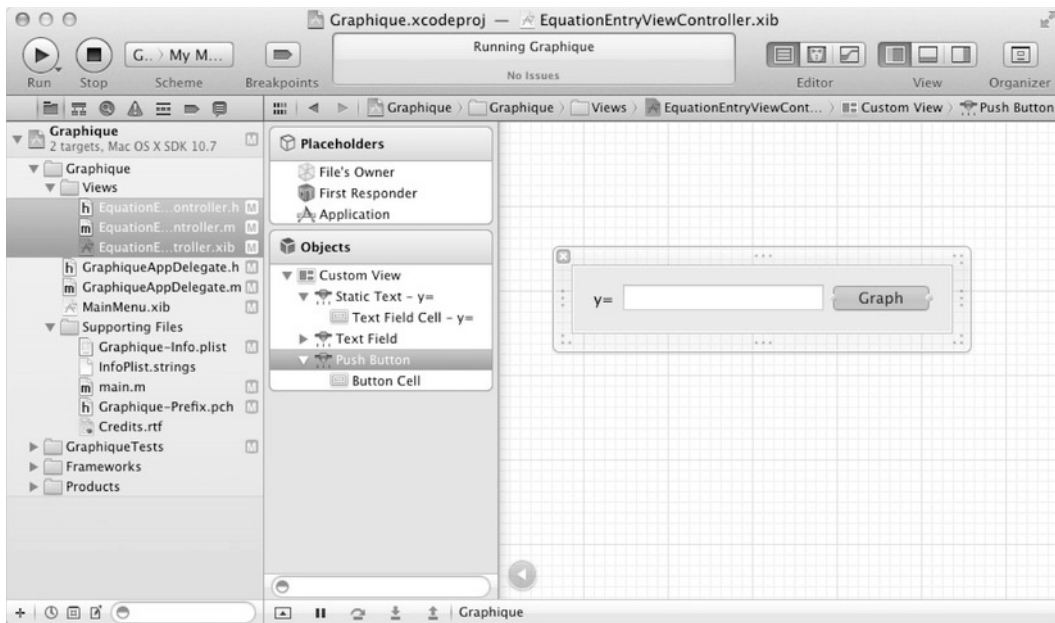


Figure 2–15. *The simple custom equation entry component*

A Primer on Automatic Reference Counting

Before moving on, you should understand what Automatic Reference Counting (ARC) is and what it means to Graphique. Introduced in Xcode 4.2, ARC eases some of the programming burden around memory management. Before the appearance of ARC, programmers dealt with a flood of program crashes and other issues related to how they handled allocating, retaining, releasing, and deallocating objects. Objective-C uses a retention count mechanism for determining whether to deallocate the memory for a given object, so the typical usage pattern for an object is as follows:

1. Allocate the object in memory.
2. Increment its retention count by sending it a retain message.
3. When done using the object, decrement its retention count by sending it a release message.
4. If the retention count reaches zero, deallocate the object.

Though the pattern is straightforward, its execution often isn't. Several parts of the code in an application, including framework code not visible to the programmer, can retain objects. Programmers can neglect to send proper retain and release messages or can incorrectly assume that objects are being retained when they aren't, or vice versa. Also, objects can create cyclical retention graphs, preventing proper releases from occurring.

When an object doesn't receive enough release messages to decrement its retention count to zero, it's never deallocated, and memory leaks. When an object is used after its

retention count reaches zero, the object has been deallocated and no longer owns that memory, causing the application to crash.

ARC wrests the burden of memory management from programmers and gives it instead to the compiler. At compile time, ARC determines the proper places to insert retain and release messages so that memory for all objects is properly managed. If you, as a programmer, try to sneak your own retain or release calls into your code, ARC refuses to compile your code until you remove those calls.

One other aspect of memory management you should understand: the difference between strong pointers and weak pointers. Strong pointers are retained, whereas weak pointers aren't. You'll see in the Graphique code throughout the book both types of pointers. Use weak pointers when you don't need to retain an object, usually because the object's lifetime exceeds the lifetime of the object using it, and when retaining it could cause a cyclical retention graph.

For more information on ARC, see its documentation at <http://clang.llvm.org/docs/AutomaticReferenceCounting.html>.

Using IBOutlet

Of course, we've now created a custom component, but we haven't used it anywhere, so it would never show if we ran the application. To use it, we simply need to add it to the right split view. The problem is that our application code currently has no knowledge of the split views because they were only defined in Interface Builder. We need a way to let Interface Builder feed this information into our code. Enter IBOutlet!

IBOutlet is a marker that has no runtime behavior, but it gives a way for Xcode to link user interface components to objects. We illustrate how to do this by linking the two split views we created in the previous section to objects in the code. This is a two-step process:

1. Define the attribute that will hold a pointer to the user interface component.
2. Link the widget to the attribute.

Open `GraphiqueAppDelegate.h`. Declare two new attributes of type `NSSplitView` and define them as properties just like you would do with any other non-UI property. Only this time, we add the IBOutlet attribute to the @property directive, as shown in Listing 2–1, to let Xcode know that they are meant for Interface Builder to see them.

Listing 2–1. *GraphiqueAppDelegate.h*

```
#import <Cocoa/Cocoa.h>

@interface GraphiqueAppDelegate : NSObject <NSApplicationDelegate>

@property (strong) IBOutlet NSWindow *window;
@property (weak) IBOutlet NSSplitView *horizontalSplitView;
@property (weak) IBOutlet NSSplitView *verticalSplitView;

@end
```

Now open `GraphiqueAppDelegate.m` to automatically generate the property accessors for the two new properties with the `@synthesize` directive:

```
@synthesize horizontalSplitView;
@synthesize verticalSplitView;
```

Now you are ready for the second step. Open `MainMenu.xib` and expand the Window - Graphique tree in the Document Outline to be able to see both split views. Select the Graphique App Delegate object and Ctrl+drag to the horizontal split view (the highest in the hierarchy). Let go of the mouse button and select `horizontalSplitView`. Do the same for the `verticalSplitView` instance, connecting it to the other split view. This step links the split views to the appropriate attributes in the `GraphiqueAppDelegate` class. Any time you access these attributes in your code, you will indeed be referring to the UI component you laid out in Interface Builder.

Hooking Up the New Component to the Application

Now that our code knows how to refer to the split views, you can add your custom equation entry component to the main window. Open `GraphiqueAppDelegate.h` and add a property for the equation entry component, as shown in Listing 2-2.

Listing 2-2. *Adding an Equation Entry Component Property*

```
#import <Cocoa/Cocoa.h>

@class EquationEntryViewController;

@interface GraphiqueAppDelegate : NSObject <NSApplicationDelegate>

@property (strong) IBOutlet NSWindow *window;
@property (weak) IBOutlet NSSplitView *horizontalSplitView;
@property (weak) IBOutlet NSSplitView *verticalSplitView;
@property (strong) EquationEntryViewController *equationEntryViewController;

@end
```

Open `GraphiqueAppDelegate.m` and add an import for `EquationEntryViewController.h`:

```
#import "EquationEntryViewController.h"
```

Remove the three `NSLog` calls that you added in Chapter 1, leaving the methods that contained them. Add a `@synthesize` directive for the `equationEntryViewController` property. Now, go to the `applicationDidFinishLaunching:` method, allocate the instance of the custom view controller, and indicate that you want it to be initialized from the XIB file using the `initWithNibName:bundle:` method:

```
self.equationEntryViewController = [[EquationEntryViewController alloc]
initWithNibName:@"EquationEntryViewController" bundle:nil];
```

NOTE: Remember that XIB files and NIB files are two different file formats to represent the same thing. NIBs were here first, and therefore you will notice that method names always refer to NIBs. Whenever they do, it's usually OK to specify a XIB file.

Now that we have an initialized controller, we can add it to the second spot in the vertical split view:

```
[self.verticalSplitView replaceSubview:[self.verticalSplitView subviews]
objectAtIndex:1] with:equationEntryViewController.view];
```

NOTE: The subviews of each split view are organized in left-to-right order for vertical split views and top-to-bottom order for the horizontal split views. The first subview is at index zero.

The complete method should look like Listing 2–3.

Listing 2–3. *Adding the Equation Entry View to the Application*

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{
    self.equationEntryViewController = [[EquationEntryViewController alloc]
initWithNibName:@"EquationEntryViewController" bundle:nil];
    [self.verticalSplitView replaceSubview:[self.verticalSplitView subviews]
objectAtIndex:1] with:equationEntryViewController.view];
}
```

Launch the application, and you should see something like Figure 2–16.

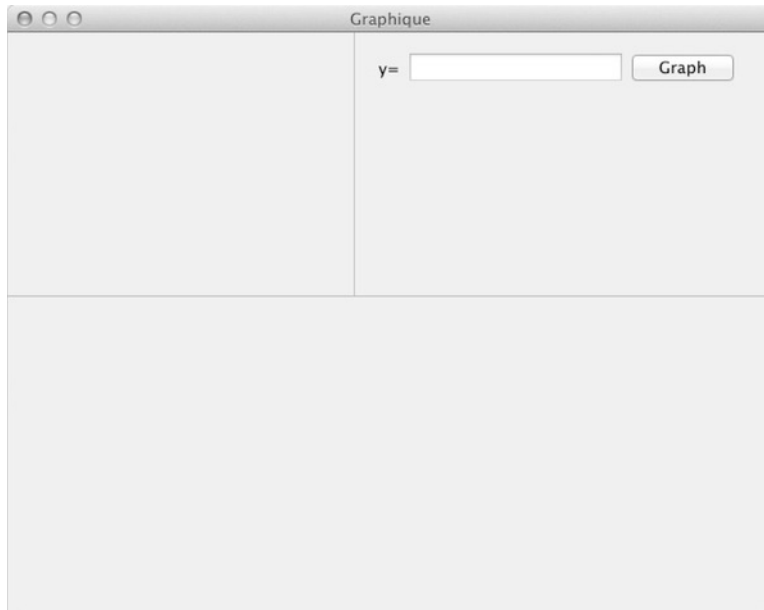


Figure 2–16. *The Graphique application with a custom equation entry component*

Resizing the Views Automatically

Split views can be resized by dragging the separator. Resize the vertical split pane, and notice how the equation editor just slides to follow the separator. This is a reasonable behavior, but it is not optimum because we are wasting space. When resizing a split pane, users usually expect one side to contract and the other side to expand to fill the available space if possible. Select `EquationEntryViewController.xib`, select the Text Field, and open the Size inspector, as illustrated in Figure 2-17.

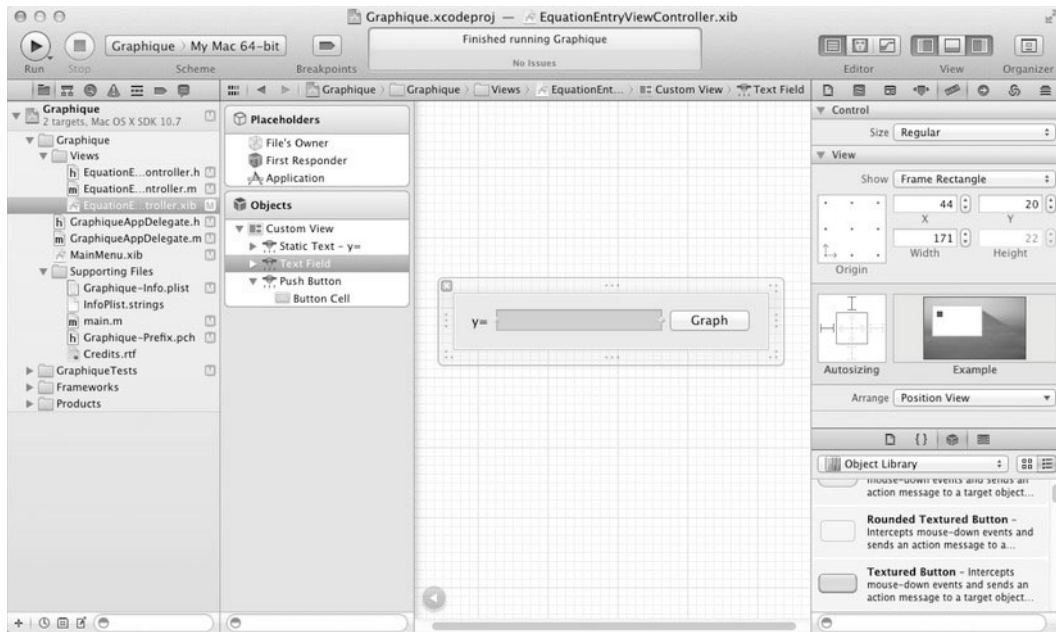


Figure 2-17. *The text field default size settings*

The Size inspector has two sections—Control and View—that allow you to specify the size of the component. In the Control section, you can pick a smaller size for each control. You have the choice among Regular, Small, or Mini. This is particularly useful if you have limited room on your user interface. The View section is of particular importance to manage automatic component resizing.

The autosizing schema shows how the component behaves as its parent component changes size. The schema shows two nested rectangles. The inner rectangle represents the selected component, the Text Field component in this case. The outer rectangle represents the parent component. Inside the inner component are two two-headed arrows, one running vertically and one running horizontally. If an arrow is enabled, this component resizes in that direction to expand or contract when the parent component changes size. The links between the outer rectangle and the inner rectangle are anchors. In this case, the Text Field component is anchored to the top-left corner of the parent component and is not set to resize. This means it will simply slide to follow the top-left corner of its parent component.

The inner-rectangle arrows are often referred to as *springs*, and earlier versions of Interface Builder depicted them as springs. The outer-rectangle anchors are often called *struts*. We use the terms *springs* and *struts* in this book.

The desired behavior is to have the label follow the left side of the parent, make the Graph button follow the right side of the parent, and have the Text Field component fill the available space in between. Since it makes no sense to expand vertically, we anchor all the components to the top boundary of the parent container.

Select the Label component, and set the autosizing schema to anchor to the top left with no resizing, as shown in Figure 2–18. It may already be set this way, so you may not have to make any changes.

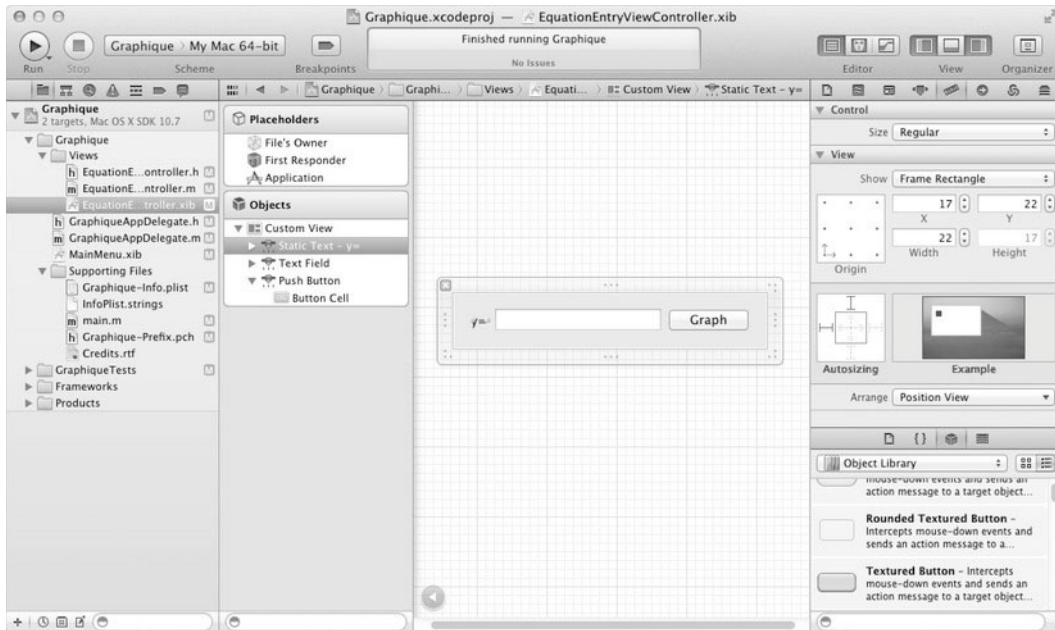


Figure 2–18. Anchoring a component to the top-left corner

Next, select the Graph button and anchor it to the top-right of its parent, as illustrated in Figure 2–19.

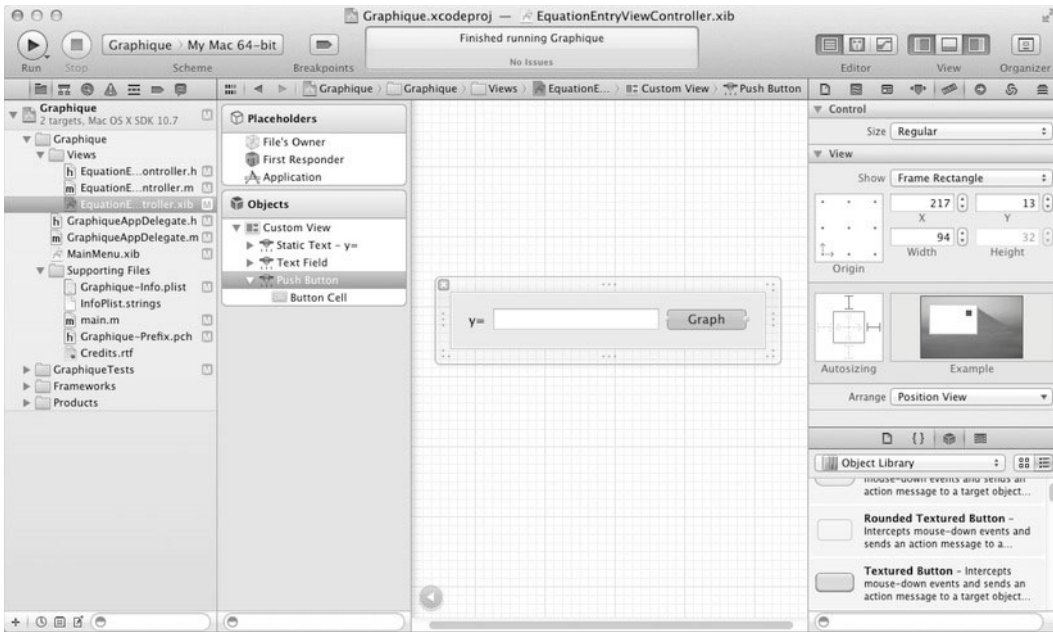


Figure 2-19. Anchoring a component to the top-right corner

Finally, select the Text Field component, and set its resizing strategy to expand horizontally to match Figure 2-20.

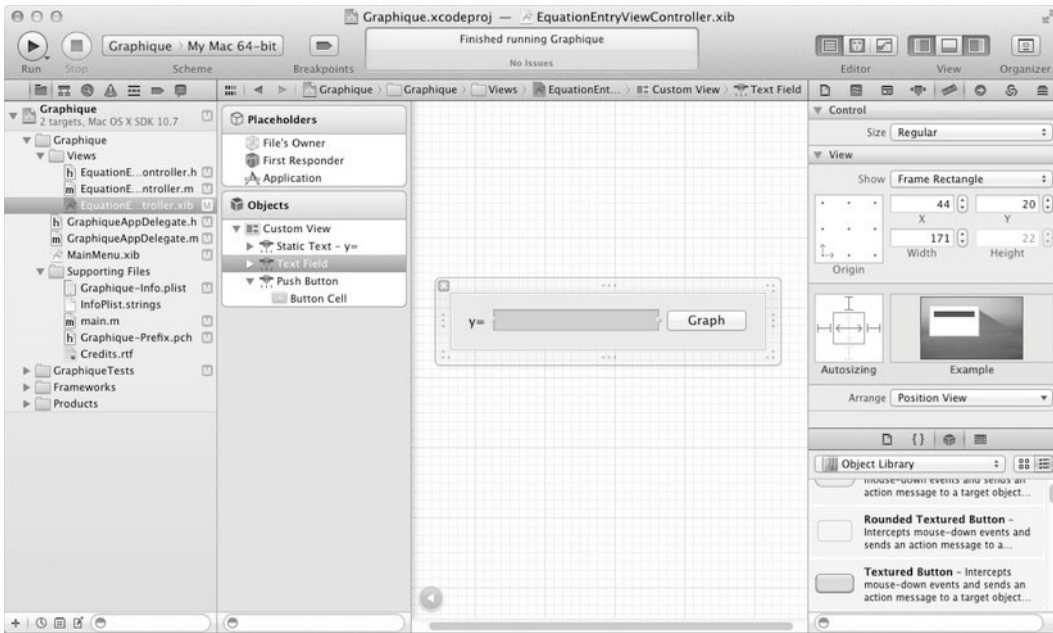


Figure 2-20. Automatically resizing to fill the available horizontal space

Launch the app and play around with the split pane separator. Notice how the equation editor resizes nicely to occupy the available space. If you size it too small, the display goes somewhat awry, but we'll fix that in Chapter 3.

Further Customizing the Components

Most Cocoa components can be customized via the Attributes inspector tab. To customize the text field in our custom view, select it and go to the Attributes inspector. In the Placeholder String field, type **2*x+1**, as shown in Figure 2–21, to have the text field display a sample equation to users so they'll have a hint for what they should type in that field.

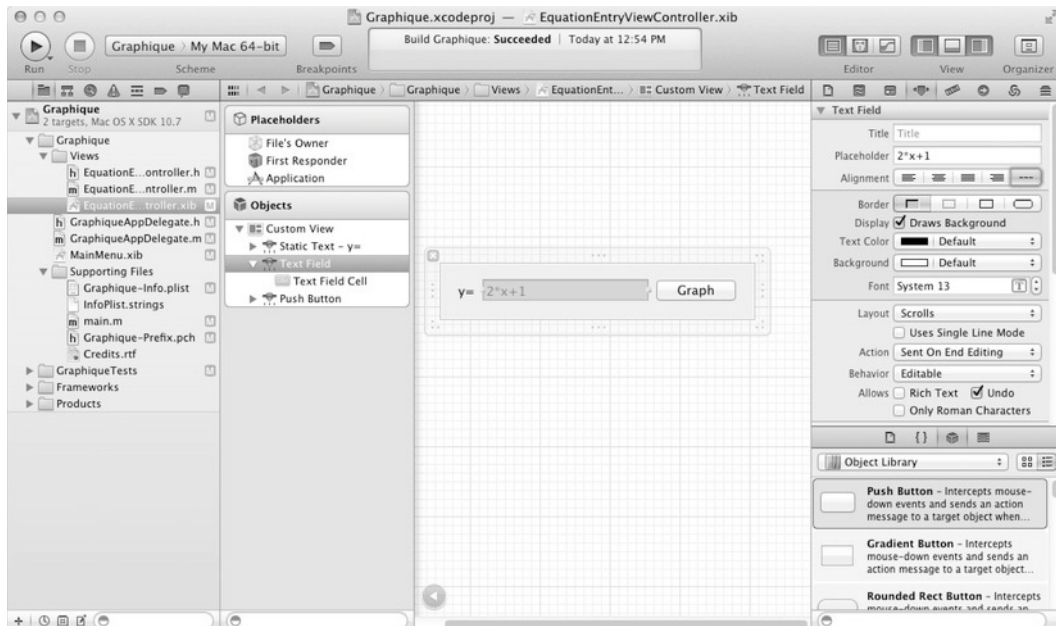


Figure 2–21. Specifying placeholder text

The placeholder string of a text field is text shown with a lighter color when nothing has been typed in the text field yet. It is used to give the user a hint of what is expected. Here, we want to show that the user is expected to type a function of x .

Launch the application again and see how the placeholder string behaves. It is shown before anything is typed, as illustrated in Figure 2–22, but disappears as soon as text is entered.

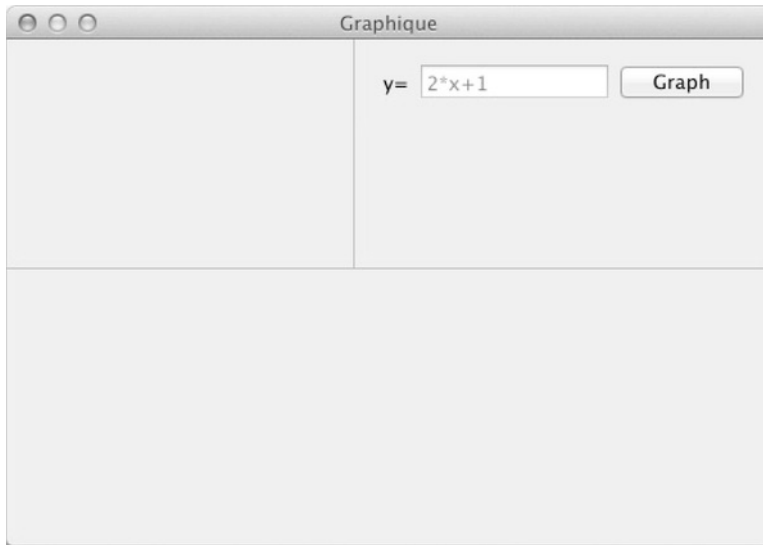


Figure 2–22. *Graphique with the equation editor*

Now you have a place for users to enter equations. Read the next section to add a place to display the results of an equation.

Creating the Graph Panel

We’re stretching the truth a little when we call this next view a “graph” panel, because we’re not yet ready to draw graphs in Graphique. Instead, we’re going to create a view for displaying the equation’s data as a table, with the domain in one column and the range in another. We also add a slider to allow people to adjust the domain interval that the table view displays. Once again, we’re going to create a custom view but this time with two controls: the table view and the slider.

To begin, create a new class in the Views group, as you did in the previous section, called `GraphTableViewController`. As with the `EquationEntryViewController` class, make it an Objective-C class and a subclass of `NSViewController`. Xcode generates the three files you need: the header file, the implementation file, and the XIB or NIB file. Select `GraphTableViewController.xib` to open it in Interface Builder. You should see a view that says Custom View but is otherwise empty, as shown in Figure 2–23.

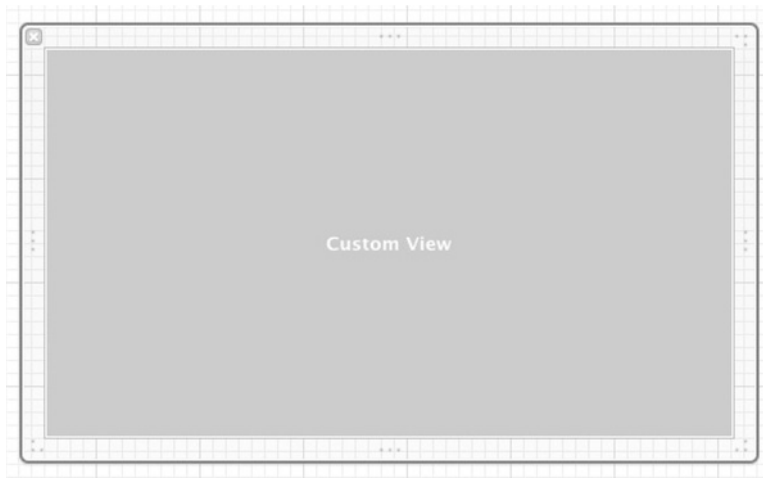


Figure 2-23. *The empty Graph View*

Adding the Horizontal Slider

That empty canvas begs for you to splash some art onto it, which you'll next do by dragging a Horizontal Slider object from the Object Library to the empty custom view. Drop it near the top of the view, using the helpful blue guides to position it, and make it span the width of the view. It should look like Figure 2-24.

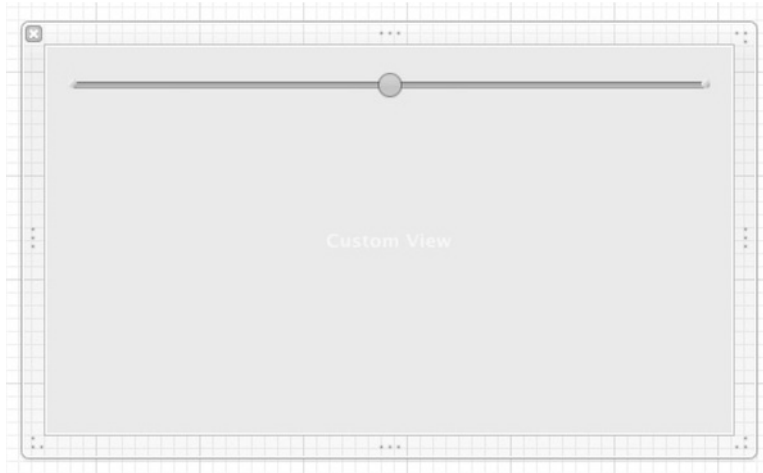


Figure 2-24. *The view with a Horizontal Slider object added*

We need to make some adjustments to the slider. First, we'll make the slider resize itself to span this view any time the view resizes, which happens when the user resizes the application window. With the horizontal slider selected, show the Size inspector. In the View section, find the Autosizing control and select the left, top, and right struts to anchor the slider to the left, top, and right of the view. Also, click the horizontal arrow

inside the box (the *spring*) to make the slider stretch to fill the view. The Autosizing control should look like Figure 2–25.

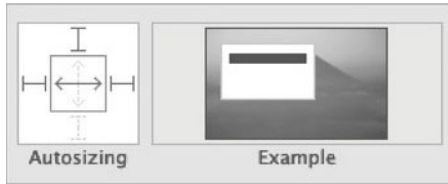


Figure 2–25. The autosizing configured for the horizontal slider

You also want to constrain the range of the horizontal slider to numbers that make sense for a domain interval. With the slider still selected, show the Attributes inspector, and in the Slider section, change the number of tick marks to 20. Set the minimum value to 0.10 and the maximum to 5.00, with the current set to 2.50. The Slider section should match Figure 2–26, and your view should look like Figure 2–27.

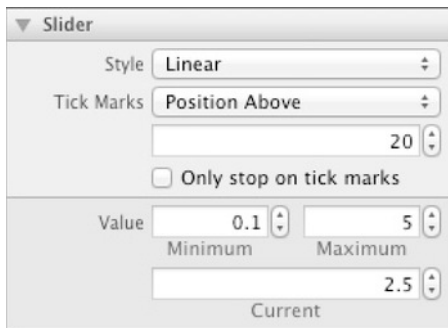


Figure 2–26. Constraining the horizontal slider

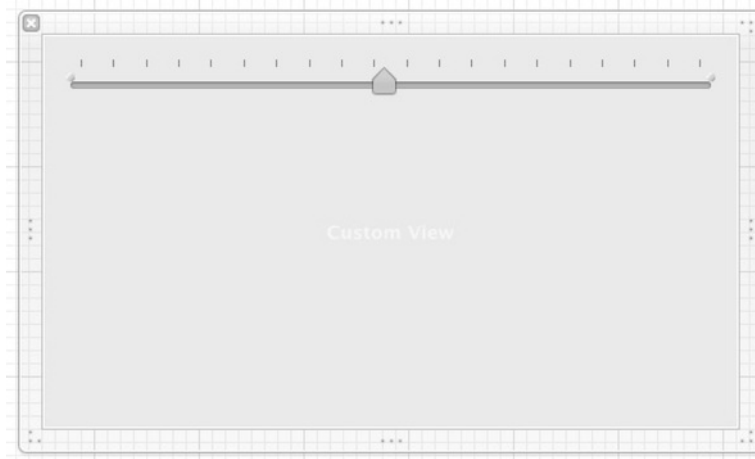


Figure 2–27. The horizontal slider after applying the constraints

With the slider in place and configured, we're halfway done creating the Graph View. In the next section, we add the table to complete the view.

Adding the Table View

Adding a table view to our custom view begins by dragging a Table View instance from the Object Library and dropping it on the view. Again, use the helpful blue lines to position and size it below the horizontal slider and to fill the rest of the view. Your view should look like Figure 2–28 when you're done.

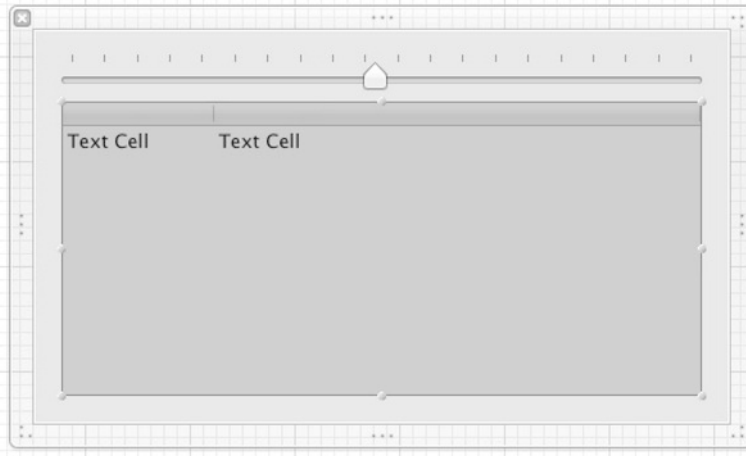


Figure 2–28. *The view after adding a table view*

The table looks perfectly positioned and sized now, but if the user were to resize the view by resizing the application window, the table would stay the same size and would no longer fill the part of the view allotted to it. To fix that, show the Size inspector, go to the Autosizing widget, and select the top, left, bottom, and right struts, as well as both springs. The Autosizing widget should match Figure 2–29.

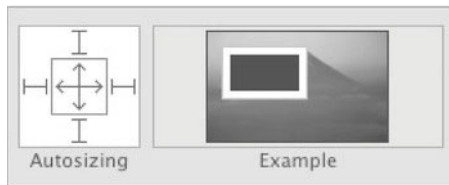


Figure 2–29. *The Autosizing settings for the table view*

Our custom view layout is complete. Next, we need to add it to the split view, which we do in the next section.

Adding the Graph Panel to the Application

To add the graph panel to the application, follow the pattern you used with the equation entry panel. This time, we want to swap it for the bottom view of the horizontal split view. Open `GraphiqueAppDelegate.h` and add a property for a `GraphTableViewController` instance, as shown in Listing 2–4.

Listing 2–4. Adding a `GraphTableViewController` Instance

```
#import <Cocoa/Cocoa.h>

@class EquationEntryViewController;
@class GraphTableViewController;

@interface GraphiqueAppDelegate : NSObject <NSApplicationDelegate>

@property (strong) IBOutlet NSWindow *window;
@property (weak) IBOutlet NSSplitView *horizontalSplitView;
@property (weak) IBOutlet NSSplitView *verticalSplitView;
@property (strong) EquationEntryViewController *equationEntryViewController;
@property (strong) GraphTableViewController *graphTableViewController;

@end
```

Go to `GraphiqueAppDelegate.m`, add an import statement for `GraphTableViewController.h`, and add a `@synthesize` directive for the `graphTableViewController` property. Edit the `applicationDidFinishLaunching:` method to create a `GraphTableViewController` instance and swap it for the horizontal split view's bottom view. The method should look like Listing 2–5.

Listing 2–5. Adding the Graph Table View to the Application

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{
    self.equationEntryViewController = [[EquationEntryViewController alloc]
initWithNibName:@"EquationEntryViewController" bundle:nil];
    [self.verticalSplitView replaceSubview:[self.verticalSplitView subviews]
objectAtIndex:1] with:equationEntryViewController.view];

    self.graphTableViewController = [[GraphTableViewController alloc]
initWithNibName:@"GraphTableViewController" bundle:nil];
    [self.horizontalSplitView replaceSubview:[self.horizontalSplitView subviews]
objectAtIndex:1] with:graphTableViewController.view];
}
```

Build and run `Graphique`. You should see a window that looks like Figure 2–30. If you resize it, you should see the content in the custom views resize to fill their allotted spaces, as in Figure 2–31.

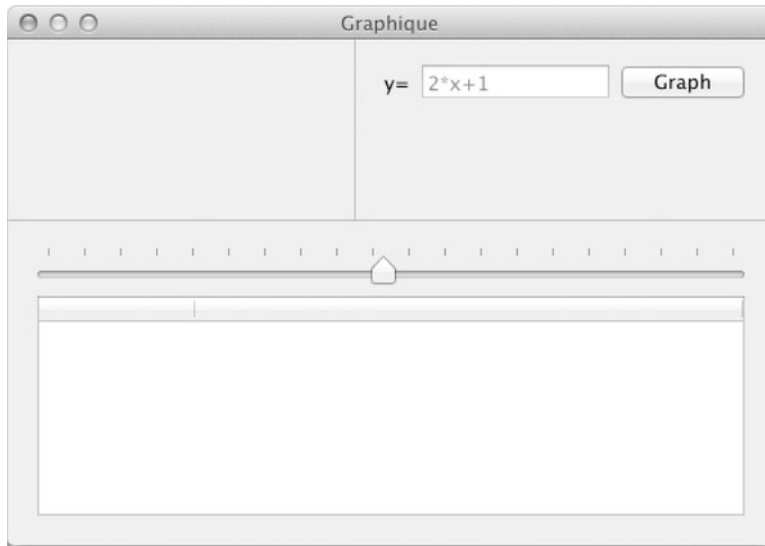


Figure 2-30. *Graphique with the graph panel added*

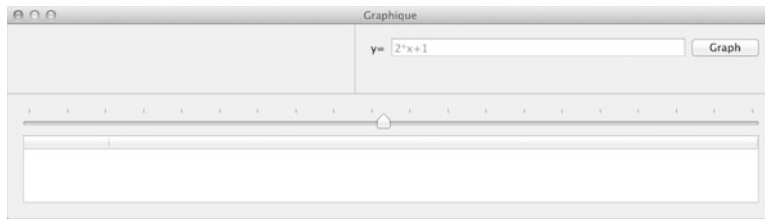


Figure 2-31. *Graphique after resizing the window*

The layout of the Graphique application is progressing. In the next section, we add the final panel, which shows the equations the user has already entered, to the split views.

Creating the Table of Recently Used Equations

The last major component of the application is the recently used equations. It provides a way for the user to get back to the equations that have been used in the past. We use this component to detail how to use the outline view, the Mac OS X implementation of a hierarchical tree view.

Just like we did for the other custom components, we create a new `NSViewController` subclass called `RecentlyUsedEquationsViewController`. This process creates the three new files you've come to expect. Figure 2-32 shows what you should see in Xcode.

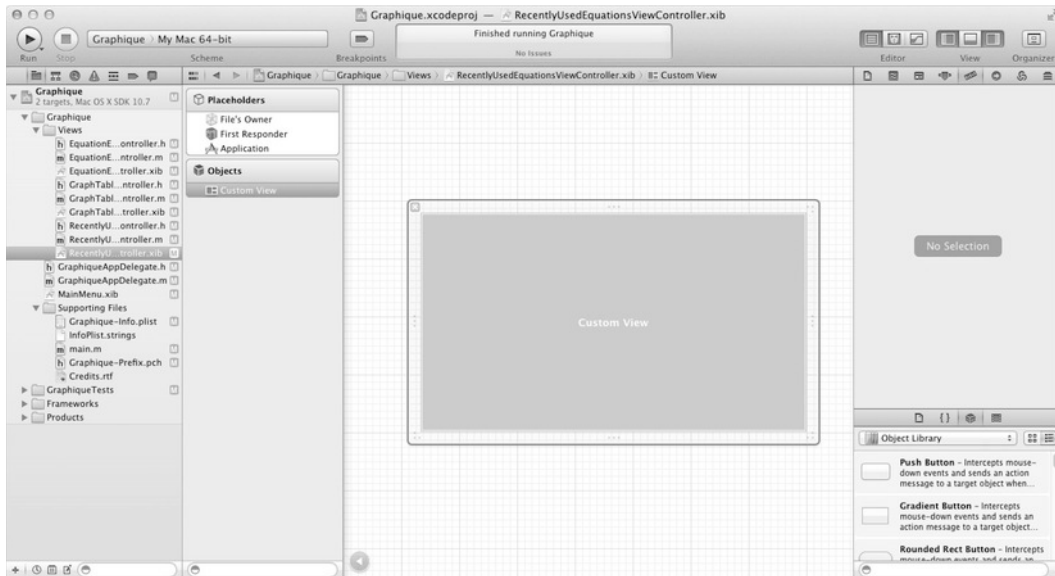


Figure 2-32. *The newly added RecentlyUsedEquationsViewController files*

Open `RecentlyUsedEquationsViewController.xib` and add an Outline View component from the standard component library. Since we don't want to see the header, highlight the Outline View object, and deselect the Headers check box on the Attributes inspector. We also want to see only one column—the hierarchy—so change the number of Columns to one, as shown in Figure 2-33.

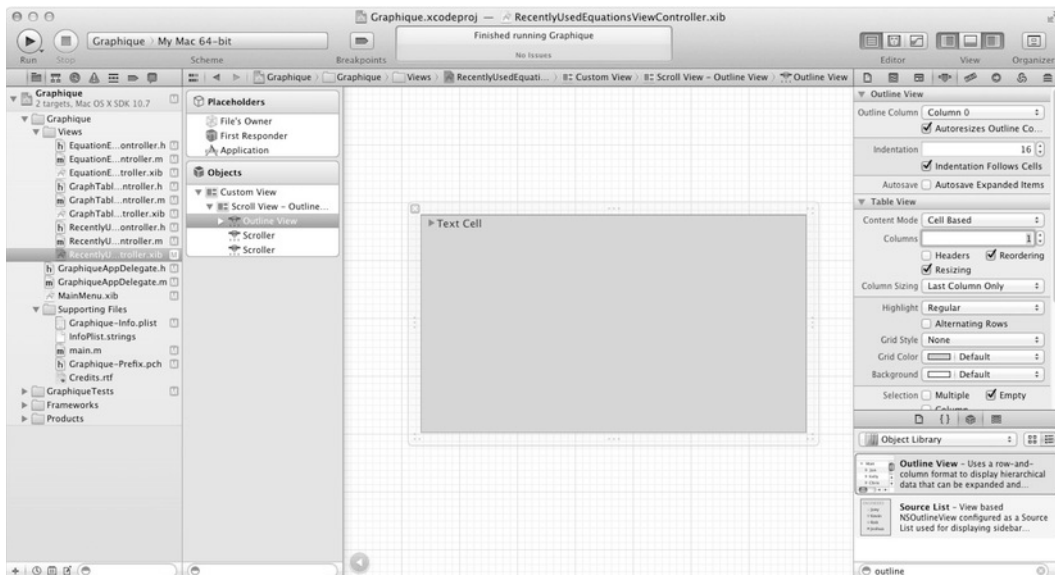


Figure 2-33. *RecentlyUsedEquationsViewController with an NSOutlineView added*

Select the Outline View component's parent, the Scroll View component right above it in the Document Outline. Open the Size inspector and select all the springs and struts so that the outline view fills the view, even after resizing, as you did with the graph table view, as shown in Figure 2–34.

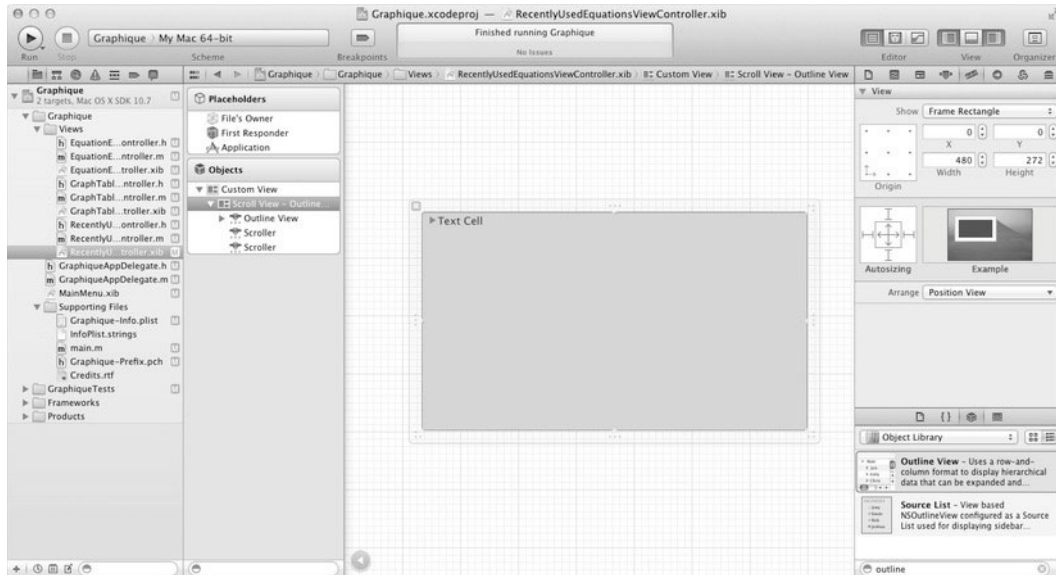


Figure 2–34. The Outline View component set to fill its parent view

As you did for the other two views you added, open `GraphiqueAppDelegate.h` and add a property for the recently used equations component, as shown in Listing 2–6.

Listing 2–6. Adding a Recently Used Equations Property

```
#import <Cocoa/Cocoa.h>

@class EquationEntryViewController;
@class GraphTableViewController;
@class RecentlyUsedEquationsViewController;

@interface GraphiqueAppDelegate : NSObject <NSApplicationDelegate>

@property (strong) IBOutlet NSWindow *window;
@property (weak) IBOutlet NSSplitView *horizontalSplitView;
@property (weak) IBOutlet NSSplitView *verticalSplitView;
@property (strong) EquationEntryViewController *equationEntryViewController;
@property (strong) GraphTableViewController *graphTableViewController;
@property (strong) RecentlyUsedEquationsViewController
*recentlyUsedEquationsViewController;

@end
```

Complete the task by opening `GraphiqueAppDelegate.m` to attach the new component to the user interface. Import `RecentlyUsedEquationsViewController.h`, add the

@synthesize directive for `recentlyUsedEquationsViewController`, and edit the `applicationDidFinishLaunching:` method to look like Listing 2–7.

Listing 2–7. *Adding the Recently Used Equations View to the Application*

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{
    self.equationEntryViewController = [[EquationEntryViewController alloc]
initWithNibName:@"EquationEntryViewController" bundle:nil];
    [self.verticalSplitView addSubview:[self.verticalSplitView subviews]
objectAtIndex:1] with:equationEntryViewController.view];

    self.graphTableViewController = [[GraphTableViewController alloc]
initWithNibName:@"GraphTableViewController" bundle:nil];
    [self.horizontalSplitView addSubview:[self.horizontalSplitView subviews]
objectAtIndex:1] with:graphTableViewController.view];

    self.recentlyUsedEquationsViewController = [[RecentlyUsedEquationsViewController
alloc] initWithNibName:@"RecentlyUsedEquationsViewController" bundle:nil];
    [self.verticalSplitView addSubview:[self.verticalSplitView subviews]
objectAtIndex:0] with:recentlyUsedEquationsViewController.view];
}
```

Outline views, just like table views, rely on a data source to provide data to display. Data sources for outline views have to comply with the `NSOutlineViewDataSource` protocol, which provides methods for describing the data to display. In this chapter, we want to show only how to display sample data, and we do not concern ourselves with how to retrieve it dynamically.

Our implementation will rely on two types of objects to represent the hierarchy. An equation node will represent one of the recently used equations, and a group node will be a node containing other nodes (think of it as a folder). We aren't yet defining the grouping criteria; this will come later when we put real data into the component.

Create a new class called `EquationItem`, a subclass of `NSObject`, with a method to return its text representation, as shown in Listing 2–8 and Listing 2–9.

Listing 2–8. *EquationItem.h*

```
#import <Foundation/Foundation.h>

@interface EquationItem : NSObject

- (NSString *)text;

@end
```

Listing 2–9. *EquationItem.m*

```
#import "EquationItem.h"

@implementation EquationItem

- (NSString *)text
{
    return @"2*x+4";
}
```

```

- (NSInteger)numberOfChildren
{
    return 0;
}

```

```
@end
```

For now, the equation is hard-coded, and an equation has no children. It is a leaf node in our hierarchy.

Now, create the `GroupItem` class to represent a group node, as shown in Listing 2–10 and Listing 2–11.

Listing 2–10. *GroupItem.h*

```

#import <Foundation/Foundation.h>

@interface GroupItem : NSObject
{
    @private
    NSString *name;
    NSMutableArray *children;
}

@property (nonatomic, retain) NSString *name;

- (NSInteger)numberOfChildren;
- (id)childAtIndex:(NSUInteger)n;
- (NSString *)text;

- (void)addChild:(id)childNode;

@end

```

Listing 2–11. *GroupItem.m*

```

#import "GroupItem.h"

@implementation GroupItem

@synthesize name;

- (id)init
{
    self = [super init];
    if (self)
    {
        children = [[NSMutableArray alloc] init];
    }
    return self;
}

- (void)addChild:(id)childNode
{
    [children addObject:childNode];
}

```

```

- (NSInteger)numberOfChildren
{
    return children.count;
}

- (id)childAtIndex:(NSUInteger)n
{
    return [children objectAtIndex:n];
}

- (NSString *)text
{
    return name;
}

@end

```

Let's take a minute to go over the group item implementation. Just like the equation item, the group item has a method called `numberOfChildren`, but in this instance it relies on the array of children to evaluate its result. The array of children defined in the header file contains a list of subitems in the hierarchy. We build our hierarchy by creating a root node and adding items to it.

Creating the Data Source

Now that we have a data structure to hold our hierarchy, we define the data source to hook up to the outline view. Open `RecentlyUsedEquationsViewController.h` and make it implement the `NSOutlineViewDataSource` protocol, as shown in Listing 2–12. We also add a group item that represents the root node, which is going to be an invisible node holding the hierarchy together.

Listing 2–12. *RecentlyUsedEquationsViewController.h Implementing the NSOutlineViewDataSource Protocol*

```

#import <Cocoa/Cocoa.h>

@class GroupItem;

@interface RecentlyUsedEquationsViewController : NSViewController
<NSOutlineViewDataSource>
{
    @private
    GroupItem *rootItem;
}

@end

```

For the data source to work, four methods must be implemented. For each of these methods, a `nil` item value serves to represent the root of the hierarchy, which we will interpret as the root item.

The first method returns the number of children the specified method has. It has the following signature:

```
- (NSInteger)outlineView:(NSOutlineView *)outlineView numberOfChildrenOfItem:(id)item
```

The second method returns whether a given item can be expanded. Typically, this means the item has children. Its signature looks like this:

```
- (BOOL)outlineView:(NSOutlineView *)outlineView isItemExpandable:(id)item
```

The third method is responsible for returning the child of the given item at the specified index. It has this signature:

```
- (id)outlineView:(NSOutlineView *)outlineView child:(NSInteger)index ofItem:(id)item
```

Finally, the fourth method returns the text to display for the given item, in the specified column. In our case for the moment, we use only one column. It uses this signature:

```
- (id)outlineView:(NSOutlineView *)outlineView objectValueForTableColumn:(NSTableColumn *)tableColumn byItem:(id)item
```

Open `RecentlyUsedEquationsViewController.m` and edit it to match Listing 2–13.

Listing 2–13. *RecentlyUsedEquationsViewController.m Implementing the Data Source*

```
#import "RecentlyUsedEquationsViewController.h"
#import "GroupItem.h"
#import "EquationItem.h"

@implementation RecentlyUsedEquationsViewController

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self)
    {
        rootItem = [[GroupItem alloc] init];

        for (int i = 0; i < 4; i++)
        {
            GroupItem *temp = [[GroupItem alloc] init];
            temp.name = [NSString stringWithFormat:@"Group %d", i + 1];

            for (int j = 0; j < 5; j++)
            {
                EquationItem *item = [[EquationItem alloc] init];
                [temp addChild:item];
            }
            [rootItem addChild:temp];
        }
    }
    return self;
}

- (NSInteger)outlineView:(NSOutlineView *)outlineView numberOfChildrenOfItem:(id)item
{
    return (item == nil) ? [rootItem numberOfChildren] : [item numberOfChildren];
}

- (BOOL)outlineView:(NSOutlineView *)outlineView isItemExpandable:(id)item
{

```

```

    return (item == nil) ? ([rootItem numberOfChildren] > 0) : ([item numberOfChildren] >
0);
}

- (id)outlineView:(NSOutlineView *)outlineView child:(NSInteger)index ofItem:(id)item
{
    if (item == nil)
    {
        return [rootItem childAtIndex:index];
    }
    else
    {
        return [(GroupItem *)item childAtIndex:index];
    }
}

- (id)outlineView:(NSOutlineView *)outlineView objectValueForTableColumn:(NSTableColumn
*)tableColumn byItem:(id)item
{
    return (item == nil) ? @"": [item text];
}

@end

```

In the `initWithNibName:bundle:` method, we initialize our data source by hard-coding all the data. The code creates the root item and creates four groups in it named Group1, Group2, Group3, and Group4. Inside each group, we create five equations. Note that in each data source method, we use `rootItem` everywhere the item is `nil`. This concludes the implementation of our data source.

Displaying the Data

The last step in getting the recently used equations view to work is to tell the outline view which data source it is supposed to use. To set this up, open `RecentlyUsedEquationsViewController.xib`, select the Outline View object, and go to the Connections inspector. Ctrl+drag the `dataSource` connection to the File's Owner placeholder, as illustrated in Figure 2–35.

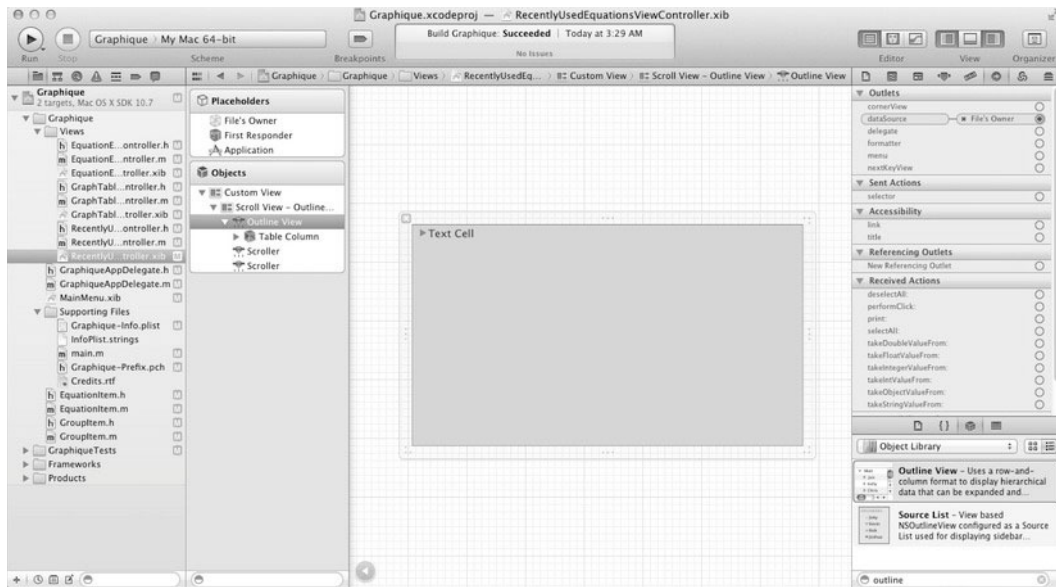


Figure 2–35. Linking the outline view data source

By default, the File's Owner of a .XIB file is the corresponding .m file. You've just linked the Outline View object's data source to the controller instance. Launch the application to make sure it all looks right. Figure 2–36 shows what you should expect.

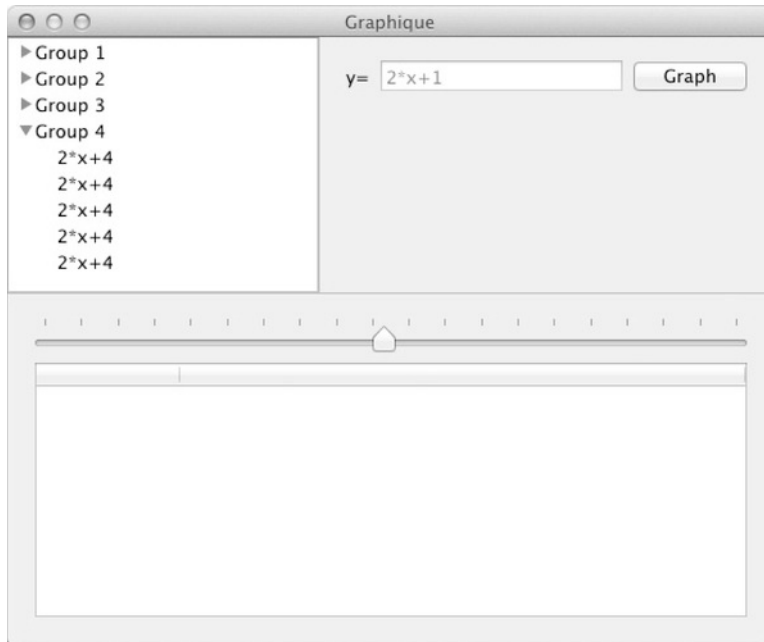


Figure 2–36. The running outline view with sample data

The split views are all filled with custom views. Many Mac OS X applications, however, offer a toolbar across the tops of their windows to provide quick access to frequently used operations. In the next section, we add a toolbar to Graphique.

Adding a Toolbar

Many applications offer a variety of tools across the top of the window, right below the title bar, represented by icons. These tools typically offer frequently used functions within the application, and providing them in the toolbar gives users quick access. Sometimes they duplicate functions that the menu provides but not always. In this section, we add a toolbar to Graphique. We don't change the stock toolbar icons yet, but we'll make changes during the course of the book as we unroll functionality for Graphique.

To add a toolbar to Graphique, select MainMenu.xib in Xcode to open it in the integrated Interface Builder, and then drag a Toolbar instance from the Object Library to the Graphique window. Drop it anywhere in the window, and it will magically jump to the top of the window and attach itself just below the title bar. Your window should look like Figure 2–37.

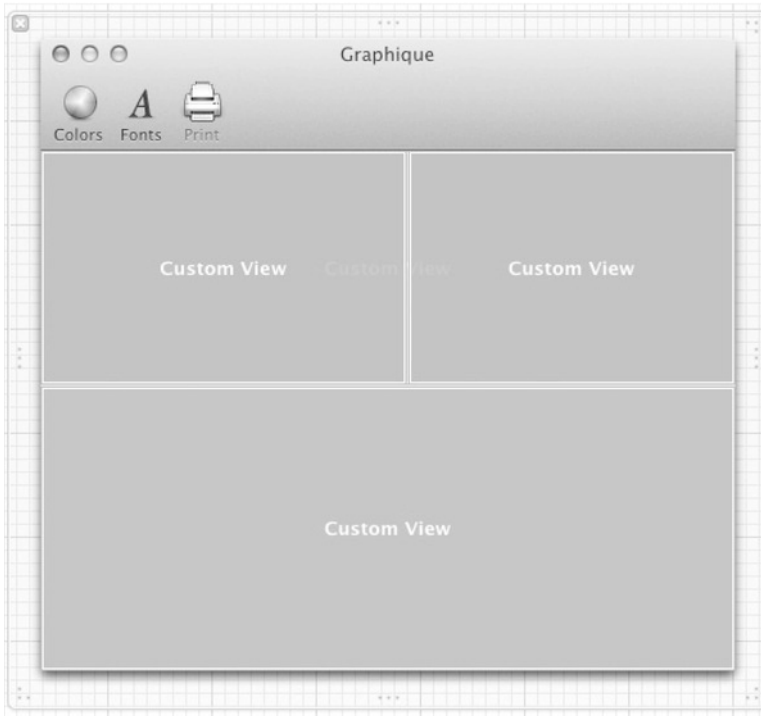


Figure 2–37. The Graphique window in Interface Builder after adding a toolbar

The stock toolbar comes with three tools: Colors, Fonts, and Print. If you build and run the application, you can see that clicking Colors brings up the standard color picker

dialog, clicking Fonts brings up the standard font selection dialog, and clicking Print does nothing, because it's disabled. At this point, we're happy with the results. As we work through the book, we'll make the toolbar actually useful.

Summary

As you worked through this chapter, you used standard Mac OS X widgets and combined them to create an attractive, resizable, and functional user interface for the Graphique application. You saw how to combine standard widgets into custom views and how to display multiple custom views in a single window using split views. You learned about table views, outline views, struts and springs, placeholder text, and other Interface Builder topics. You built a great user interface.

Looks only go so far, however. At this point, Graphique looks good and resizes properly, and you can tell it to graph equations by entering them and clicking the Graph button. Like too many teenagers, however, Graphique ignores your requests and just slumps there on your screen, doing nothing. It's time to teach Graphique some manners and have it respond to user input—that's the subject of the next chapter.

Handling User Input

In the movie *The Sound of Music*, the governess Maria (played by Julie Andrews) teaches the Von Trapp children how to sing. In one scene, as she's introducing notes to the children, she teaches them a melody with note names that, while sounding good, has lyrics composed only of the names of notes: "Sol Do La Fa Mi Do Re...." One of the children, Brigitta, responds, "But it doesn't mean anything." Maria responds, "So we put in words." Right now, Graphique doesn't mean anything. It's just a user interface. In this chapter, we put in words. We put in the functionality that makes Graphique mean something. By the end of this chapter, Graphique will look like Figure 3-1. It will allow users to enter equations and graph them textually. It will be a functional application.

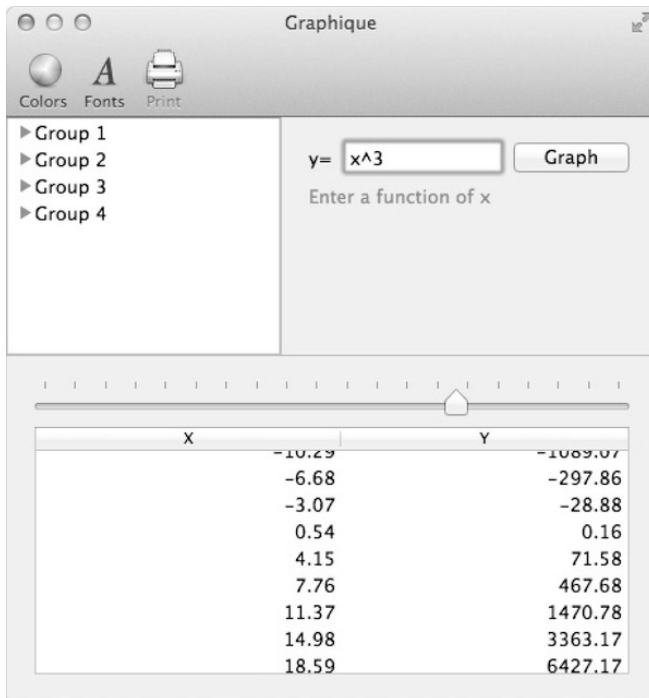


Figure 3-1. The Graphique application at the end of this chapter

Resizing the Views

Some people like their windows big, and some like them small. Some users have 30" cinema displays, and some work from 11" notebook screens. Sometimes people will want to see only the graph and hide the recent equations and the equation editor. Sometimes people will want to drag the split views around and resize the entire Graphique window. Maybe some people will want Graphique to take over their entire desktops. You can control and customize the resizing behavior of Graphique's window and views to improve how the user interacts with the application. In this section, you learn how to do that.

Resizing the Window

Open `MainMenu.xib` in Xcode and select the Graphique window. Show the Attributes inspector to see ways you can control the Graphique window's appearance and sizing capability. The Attributes inspector should match Figure 3–2.

You can see a slew of options, but the ones we focus on here are the `Resize` check box and the `Full Screen` drop-down, so locate those in the Attributes inspector. The `Resize` check box is currently checked. Uncheck that and run Graphique. The Graphique window looks as it did before, but as you move the mouse to the window sides or corners, you see that the mouse pointer no longer turns into a resizing arrow, and you can't resize the window. We obviously don't want to constrain our users to a single window size, so quit Graphique and recheck the `Resize` check box.

The other option we want to look at, `Full Screen`, is currently set to `Unsupported`. `Full Screen` support is new with Lion, but applications must explicitly specify this option to gain support for it. When you run Graphique, you see no `Full Screen` control in the upper-right corner of the window. Change this setting to `Primary Window` and run Graphique. You see that now the Graphique window has the `Full Screen` control in its upper right, which looks like two opposing arrows pointing northeast and southwest, as shown in Figure 3–3. Click this button to make Graphique fill your screen. We'll leave this setting so users can do the same when they run Graphique.

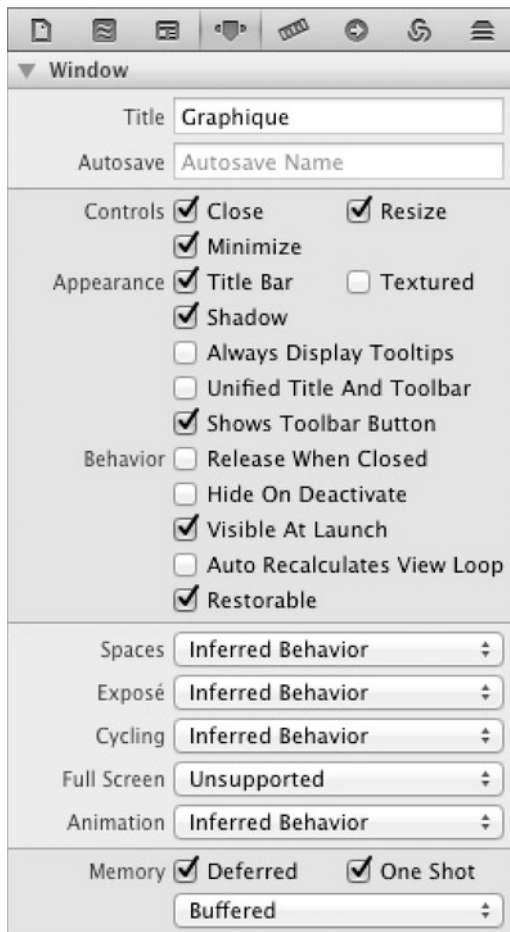


Figure 3–2. *The Attributes inspector for the Graphique window*

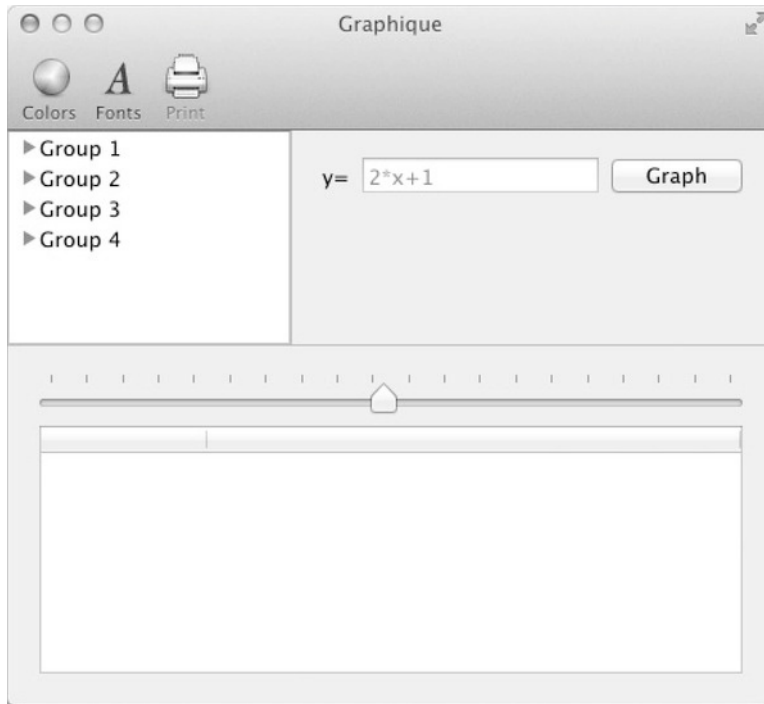


Figure 3–3. *The Graphique window with Full Screen support*

Constraining the Split View Sizes

As the Graphique application currently stands, users can resize the split views to dimensions that don't make sense. They can, for example, size the Equation Entry View so small that the text field and button no longer fit. Using the `NSSplitViewDelegate` protocol, you can control the minimum size of your split views and also whether double-clicking the splitter will collapse the view.

Constraining the Minimum Size

`NSSplitViewDelegate` has two methods for controlling the size of its views: `splitView:constrainMinCoordinate:ofSubviewAt:` and `splitView:constrainMaxCoordinate:ofSubviewAt:`. These methods receive the index of the view to the left of the divider, in the case of a vertical split view, or the index of the view above the divider, in the case of a horizontal split view. In your implementation of these methods, you return a float value that represents the minimum or maximum proposed value for the width or height of the view, as appropriate. For vertical split views, you return a width, and for horizontal split views, you return a height. For `splitView:constrainMinCoordinate:ofSubviewAt:`, you return a minimum, and for `splitView:constrainMaxCoordinate:ofSubviewAt:`, you return a maximum. To constrain the minimum width of a view to 100 pixels, for example, you'd add this method:

```

- (CGFloat)splitView:(NSSplitView *)splitView
constrainMinCoordinate:(CGFloat)proposedMinimumPosition
ofSubviewAt:(NSInteger)dividerIndex
{
    return 100.0;
}

```

To properly display the Equation Entry View, we must allow it to shrink only to 175 pixels. That allows 82 pixels for the Graph button, 30 pixels for the y= label, and 63 pixels for the text field and any padding.

Constraining the Maximum Size

Dragging a split view's divider prompts your delegate's method to be called to constrain its size. Because we want to set a minimum size for the right view (the Equation Entry View), we must prevent the divider from extending past a certain point. This means we must impose a maximum size constraint on the divider so that it does not go beyond its overall width minus the minimum width we want to impose on the Equation Entry View. To do this, make *RecentlyUsedEquationsViewController* the delegate for the vertical split view, and implement the `splitView:constrainMaxCoordinate:ofSubviewAt:` method to return a maximum size of the total width of the split view minus 175 pixels. Begin by declaring that *RecentlyUsedEquationsViewController* implements the *NSSplitViewDelegate* in *RecentlyUsedEquationsViewController.h*, as shown in Listing 3–1.

Listing 3–1. *RecentlyUsedEquationsViewController.h*

```

#import <Cocoa/Cocoa.h>

@class GroupItem.h;

@interface RecentlyUsedEquationsViewController : NSViewController
<NSOutlineViewDataSource, NSSplitViewDelegate>
{
    @private
    GroupItem *rootItem;
}

@end

```

At the top of *RecentlyUsedEquationsViewController.m*, use `#define` to avoid using a magic number:

```
#define EQUATION_ENTRY_MIN_WIDTH 175.0
```

implement the `splitView:constrainMaxCoordinate:ofSubviewAt:` method to get the overall width of the split view, subtract 175 pixels, and return the result, as shown in Listing 3–2.

Listing 3–2. Constraining the Maximum Width

```
# pragma mark - NSSplitViewDelegate methods

- (CGFloat)splitView:(NSSplitView *)splitView
  constrainMaxCoordinate:(CGFloat)proposedMinimumPosition
  ofSubviewAt:(NSInteger)dividerIndex
{
    return splitView.frame.size.width - EQUATION_ENTRY_MIN_WIDTH;
}
```

Finally, go to `GraphiqueAppDelegate.m`, find where you create the `RecentlyUsedEquationsViewController` instance in the `applicationDidFinishLaunching:` method, and set the instance as the delegate for the vertical split view, as in Listing 3–3.

Listing 3–3. Setting the `RecentlyUsedEquationsViewController` Instance as the Vertical Split View Delegate

```
recentlyUsedEquationsViewController = [[RecentlyUsedEquationsViewController alloc]
initWithNibName:@"RecentlyUsedEquationsViewController" bundle:nil];
[self.verticalSplitView replaceSubview:[self.verticalSplitView subviews]
objectAtIndex:0] with:[recentlyUsedEquationsViewController view]];
self.verticalSplitView.delegate = recentlyUsedEquationsViewController;
```

Run `Graphique` and drag the vertical divider left and right. You find that you can drag it only so far to the right before it stops, as shown in Figure 3–4, leaving enough room to render the Equation Entry View's label, text view, and button.

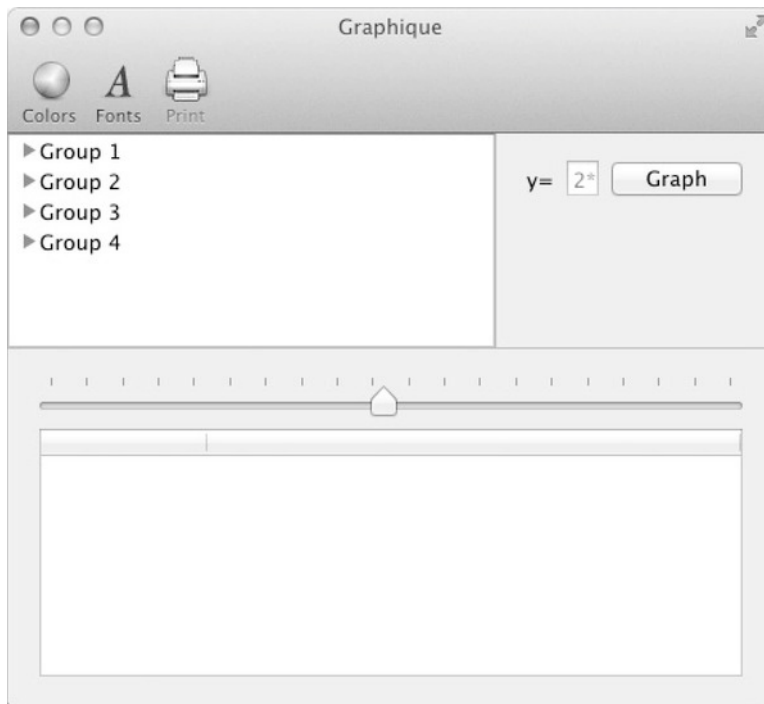


Figure 3–4. The vertical divider in the top split view moved to the right as far as it will go

Just as you're ready to check this item off your task list, however, you try resizing the window and discover that you can shrink the window smaller than 175 pixels wide, and the Equation Entry View gets crunched. The `splitView:constrainMaxCoordinate:ofSubviewAt:` method is called in response to moving the split view divider, not when resizing the window. What you must do is constrain the width of the window, which you learn how to do next.

Constraining the Window Size

To constrain the minimum overall window size and preserve the width of the Equation Entry View, you must do two things:

1. Constrain the window size.
2. Implement the `splitView:resizeSubviewsWithOldSize:` from `NSSplitViewDelegate`.

To address the first item, open `MainMenu.xib`, select Window – Graphique, and open the Size inspector. Check the box beside Minimum Size, and enter 175 for width and 200 for height, as shown in Figure 3–5.

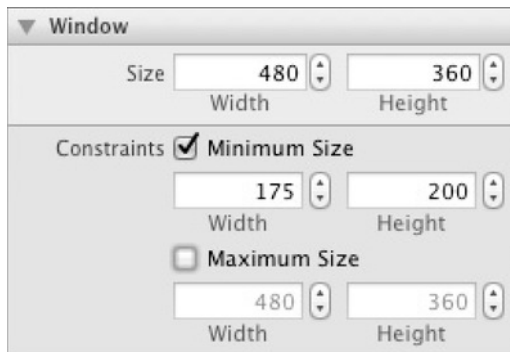


Figure 3–5. Constraining the minimum size of the window

This will set the minimum size of the window's content, ignoring any borders or other chrome, to 175 pixels wide and 200 pixels tall.

If you build and run Graphique now, however, you see that the minimum size of the window is constrained but that the split view resizes proportionally and the Equation Entry View becomes too narrow, as shown in Figure 3–6.

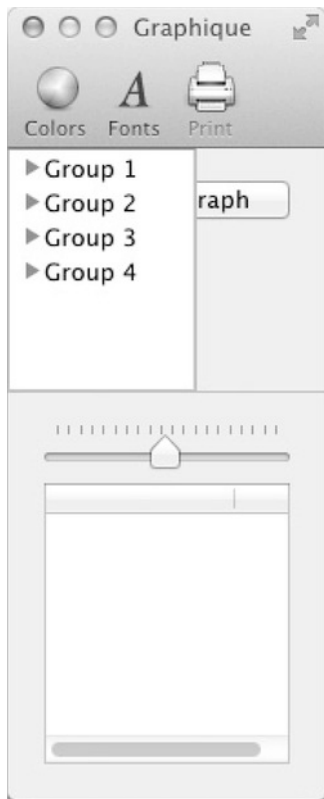


Figure 3–6. *The window size constrained but the Equation Entry View still too narrow*

When a split view resizes, it calls its delegate's `splitView:resizeSubviewsWithOldSize:` method, passing the size of the split view as it was before it was resized. In this method, you should adjust the sizes of the subviews inside the split view, taking into account the divider width. For our algorithm, we take the following approach:

- The Equation Entry View must be 175 pixels wide, less the divider width.
- As the window resizes, the Recent Equations View should stay the same size, and the Equation Entry View should grow or shrink.
- The Recent Equations View should shrink if it must to accommodate the Equation Entry View.
- We should try to make the Recent Equations View at least 100 pixels wide, as afforded by the window size and the Equation Entry View, unless it's been collapsed to zero width.

To accomplish this, add a #define for the preferred minimum width of the recently used equations view:

```
#define PREFERRED_RECENT_EQUATIONS_MIN_WIDTH 100.0
```

Next, add the code in Listing 3–4 to RecentlyUsedEquationsViewController.m.

Listing 3–4. *splitView:resizeSubviewsWithOldSize:*

```
- (void)splitView:(NSSplitView *)splitView resizeSubviewsWithOldSize:(NSSize)oldSize
{
    // Get the new frame of the split view
    NSSize size = splitView.bounds.size;

    // Get the divider width
    CGFloat dividerWidth = splitView.dividerThickness;

    // Get the frames of the recently used equations panel and the equation entry panel
    NSArray *views = splitView.subviews;
    NSRect recentlyUsed = [[views objectAtIndex:0] frame];
    NSRect equationEntry = [[views objectAtIndex:1] frame];

    // Set the widths
    // Sizing strategy:
    // 1) equation entry must be a minimum of 175 pixels minus the divider width
    // 2) recently used will stay at its current size, unless it's less than 100 pixels
    //    wide
    // 3) If recently used is less than 100 pixels, grow it as much as possible until it
    //    reaches 100
    float totalFrameWidth = size.width - dividerWidth;

    // Set recently used to the desired size (at least 100 pixels wide), or keep at zero
    // if it was collapsed
    recentlyUsed.size.width = recentlyUsed.size.width == 0 ? 0 :
    MAX(PREFERRED_RECENT_EQUATIONS_MIN_WIDTH, recentlyUsed.size.width);

    // Calculate the size of the equation entry based on the recently used width
    equationEntry.size.width = MAX((EQUATION_ENTRY_MIN_WIDTH - dividerWidth),
    (totalFrameWidth - recentlyUsed.size.width));

    // Now that the equation entry is set, recalculate the recently used
    recentlyUsed.size.width = totalFrameWidth - equationEntry.size.width;

    // Set the x location of the equation entry
    equationEntry.origin.x = recentlyUsed.size.width + dividerWidth;

    // Set the widths
    [[views objectAtIndex:0] setFrame:recentlyUsed];
    [[views objectAtIndex:1] setFrame:equationEntry];
}
```

Read through this method and its comments to understand what it's doing. This method gets the overall new size of the split view, calculates the sizes of the two views (the Equation Entry View and the Recent Equations View) according to the earlier rules, and sets the sizes into the views. Build and run Graphique and resize the window. You

should see that you can stretch the window, which stretches the Equation Entry View, as shown in Figure 3–7. You can shrink the window, which should shrink the Equation Entry View and, eventually, the Recent Equations View, as shown in Figure 3–8. Finally, you should be able to stretch the window again and see that the Recent Equations View grows to 100 pixels (if it hasn't been collapsed to zero width) and then stops expanding, as shown in Figure 3–9.

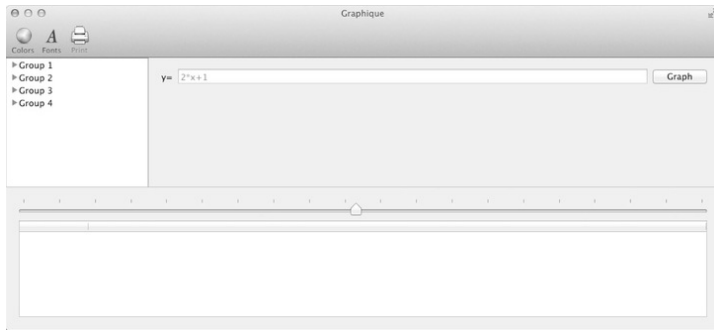


Figure 3–7. Resizing Graphique stretches the Equation Entry View



Figure 3–8. Graphique with the Recent Equations View shrunk

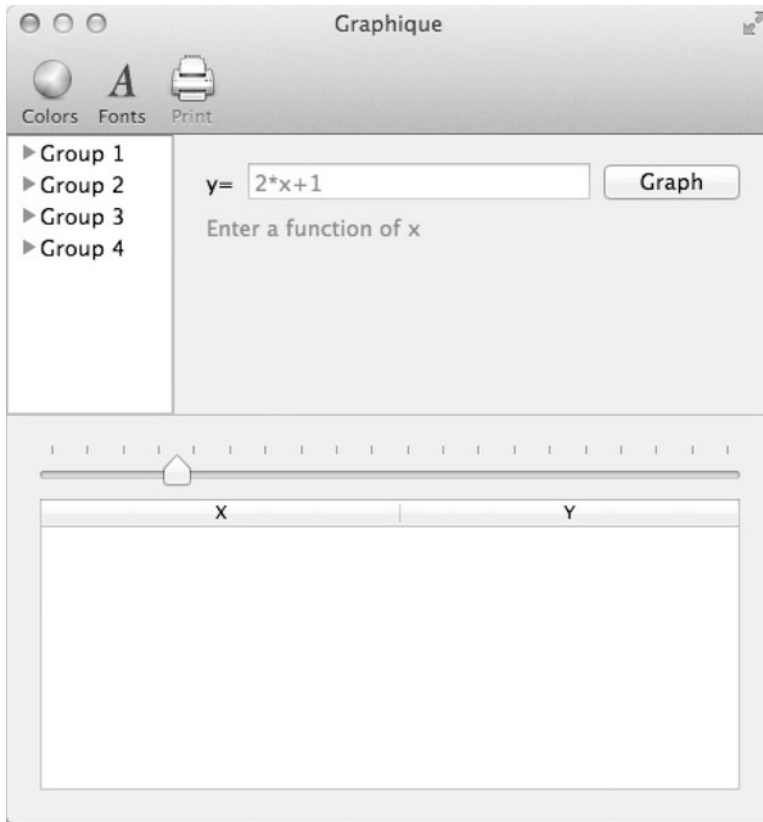


Figure 3–9. *Graphique with the Recent Equations View at 100 pixels*

Collapsing a Subview

Before we move on from handling user interaction with our split views, we want to add one more touch: if the user double-clicks the vertical divider between the Recent Equations View and the Equation Entry View, we hide (or collapse) the Recent Equations View. To do this, we implement two methods from `NSSplitViewDelegate`:

- `splitView:canCollapseSubview:` returns whether a particular split view can collapse
- `splitView:shouldCollapseSubview:forDoubleClickOnDividerAtIndex:` is called when the user double-clicks a divider

When the user double-clicks a divider, `splitView:shouldCollapseSubview:forDoubleClickOnDividerAtIndex:` is called twice, once for the first view and once for the second. You should either return `NO` for both or return `YES` for only one, because the result of returning `YES` for both is undefined. If you return `YES` for a view, and `splitView:canCollapseSubview:` also returns `YES` for that view, then the view will collapse and disappear.

In our implementation for these methods, which we place in `RecentlyUsedEquationsViewController.m`, we return YES if the specified subview is the Recent Equations View. Add the code from Listing 3–5 to `RecentlyUsedEquationsViewController.m`.

Listing 3–5. *NSSplitViewDelegate Methods to Allow Collapsing the Recent Equations View*

```
- (BOOL)splitView:(NSSplitView *)splitView shouldCollapseSubview:(NSView *)subview
forDoubleClickOnDividerAtIndex:(NSInteger)dividerIndex
{
    return subview == self.view;
}

- (BOOL)splitView:(NSSplitView *)splitView canCollapseSubview:(NSView *)subview
{
    return subview == self.view;
}
```

Build and run Graphique, and double-click the vertical divider between the Recent Equations View and the Equation Entry View, and the Recent Equations View should disappear, as shown in Figure 3–10.

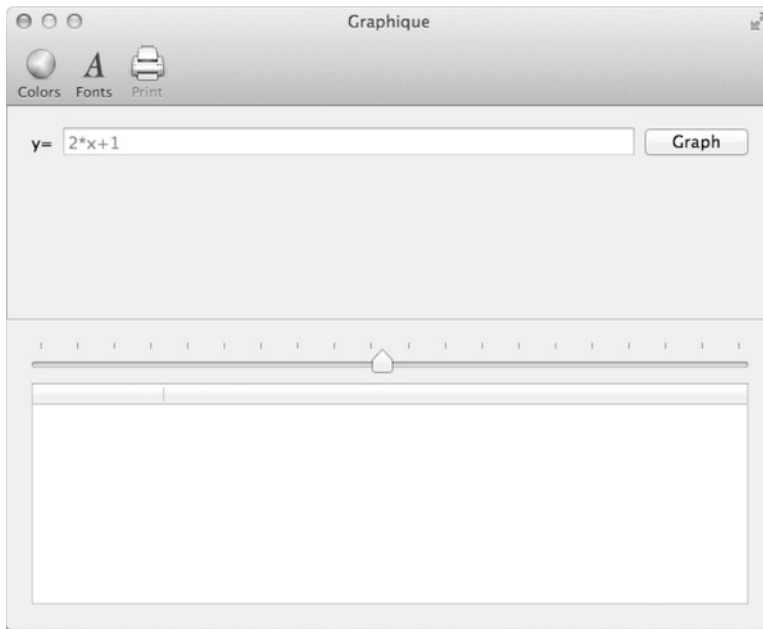


Figure 3–10. *Graphique with the Recent Equations View collapsed*

Handling window and split view resizing can be a chore to do properly, but if implemented correctly, it can create a more friendly user experience.

Handling Button Presses

In the previous chapter, you learned how to use `IBOutlet` to attach Interface Builder UI components to variables in the code. In this section, we show you how to send events and call methods in your code from UI components.

The whole idea of the application is to graph an equation curve. So far, we've built the UI components that handle inputting the equation, but we haven't yet done anything with that equation. Let's change that. We want to be able to graph an equation when the user clicks the Graph button or when the user presses the Return key when the equation text field has the focus.

The Model-View-Controller Pattern

Apple, with the Cocoa framework, has done a great job staying consistent with the classic Model-View-Controller (MVC) design pattern. In essence, the MVC pattern states that responsibilities should be divided among three major components of the application. The model holds the data. The view displays the data. The controller makes decisions based on events. Events can be triggered by changes in the view (for example, the user clicks a button), but they could also be triggered by changes in the data or by other controllers in the application. Figure 3–11 shows how the components interact.

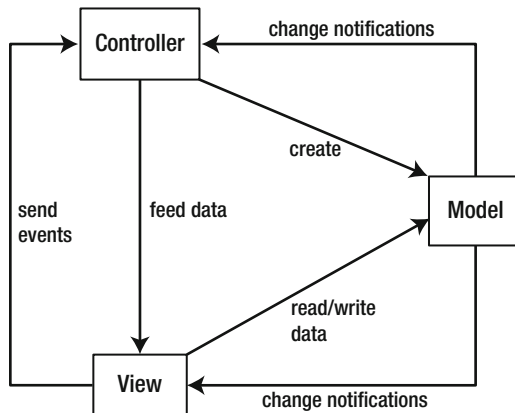


Figure 3–11. *The Model-View-Controller pattern*

Using IBAction

We are now ready to have our controller receive events. Methods that receive UI events all have a similar signature. Their return type must be `IBAction`, and they must have a single `id` argument. If you look closer at the documentation, you will notice that `IBAction` is actually defined as `void`. Similar to `IBOutlet`, `IBAction` is simply a marker that helps Interface Builder locate the method.

Since we want to receive an event from the Equation Entry View, the appropriate controller for receiving the event is `EquationEntryViewController`. Add a new method to that controller and call it `equationEntered:`. For now, we will make it print a statement in the Console when the event is activated. The method should look like as shown here:

```
- (IBAction)equationEntered:(id)sender
{
    NSLog(@"Equation entered");
}
```

Don't forget to add `- (IBAction)equationEntered:(id)sender;` to `EquationEntryViewController.h` to make the method visible outside that class.

Next, open `EquationEntryViewController.xib`, select the Graph button, and open the Connections inspector. In the connections, you will notice the Sent Actions selector entry. Click the empty circle and drag to the File's Owner placeholder. A pop-up appears and shows your new method, as shown in Figure 3-12.

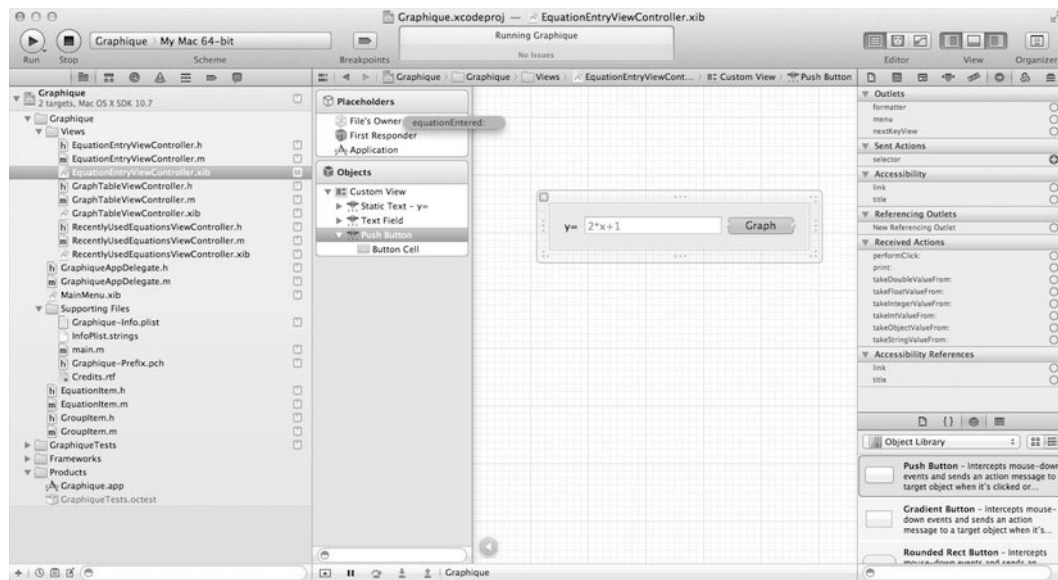


Figure 3-12. Linking a button to its action

Select the method and the link should be created, as illustrated in Figure 3-13.

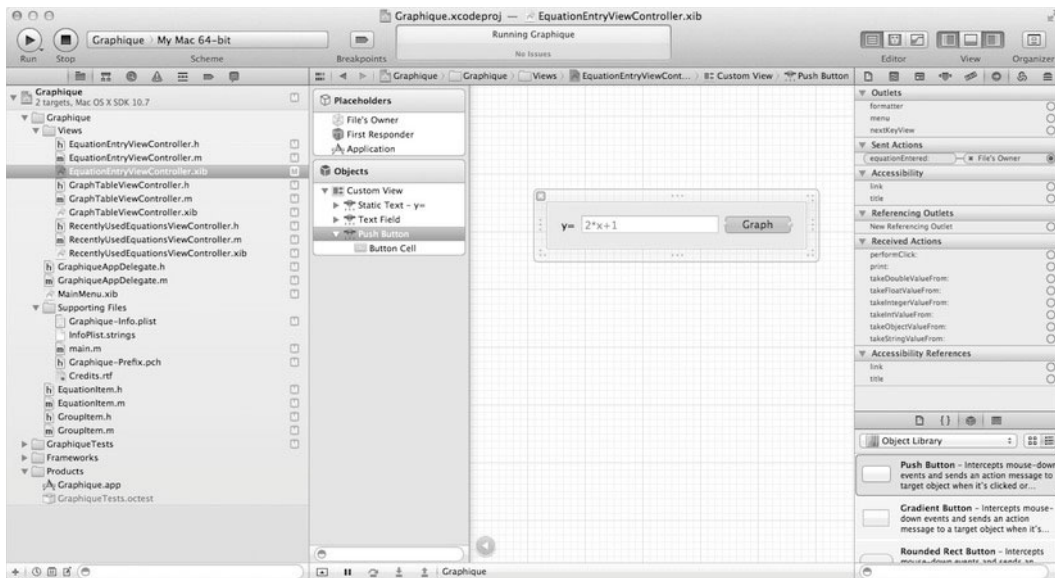


Figure 3-13. The button linked to its action in the File's Owner

Save and launch the app. Every time you click the Graph button, the `equationEntered:` method executes and prints a statement in the log console.

To wire the Return key to the same action, go back to `EquationEntryViewController.xib` in Xcode, select the text field and the Connections inspector, and link the text field's selector entry to the `equationEntered:` method so that pressing Return will fire the same notification.

NOTE: Action methods require an argument typically named `sender`. The sender is, as its name implies, the component that has emitted the notification. In our case, sender is the button instance.

Creating the Model: Equation

When users enter equations, they do so in the form of strings of characters. To keep everything clean, it is usually best to create a data structure that will hold the data and also help access the data in a controlled manner. In our case, the data structure is an equation, so let's create a new subclass of `NSObject` called `Equation` with a convenience `init:` method that will accept a string in the form of an `NSString`. Create a new Objective-C class called `Equation` by selecting **File > New > New File...** from the menu. Be sure to make it a subclass of `NSObject`.

By creating a custom model class, we hide the fact that, for now at least, an equation is simply represented as a string. Find the code for the header file, `Equation.h`, in Listing 3-6, and the code for the implementation file, `Equation.m`, in Listing 3-7.

Listing 3–6. *Equation.h*

```
#import <Foundation/Foundation.h>

@interface Equation : NSObject
{
    @private
    NSString *text;
}
@property (nonatomic, retain) NSString *text;

- (id)initWithString:(NSString*)string;

@end
```

Listing 3–7. *Equation.m with its Custom init Method*

```
#import "Equation.h"

@implementation Equation

@synthesize text;

- (id)initWithString:(NSString *)string
{
    self = [super init];
    if (self) {
        self.text = string;
    }
    return self;
}

- (NSString *)description
{
    return [NSString stringWithFormat:@"Equation [%@]", self.text];
}

@end
```

Our equation code would not be complete without a method to evaluate the equation for a given value. For simplicity, we assume that the variable is always named *x* in every equation. Edit *Equation.h* again to declare a new method:

```
- (float)evaluateForX:(float)x;
```

Implementing this method could be its own chapter. Between the use of grammars and semantic trees, there would be plenty to write. Here, we focus on implementing this method so that it is functional enough to help illustrate the rest of the examples in the book. Luckily for us, Mac OS X is based on the Unix operating system, which comes with the *awk* command. In this implementation, we leverage that command to do all the work. We don't dive into that code here; for further reading, review the *NSTask* documentation for launching programs from within your program.

Listing 3–8 shows *Equation.h* with the additional method, and Listing 3–9 shows *Equation.m* with the implementation of the method.

Listing 3–8. *Equation.h with the evaluateForX: Method*

```
#import <Foundation/Foundation.h>

@interface Equation : NSObject
{
    @private
    NSString *text;
}
@property (nonatomic, retain) NSString *text;

- (id)initWithString:(NSString *)string;
- (float)evaluateForX:(float)x;

@end
```

Listing 3–9. *Equation.m with the evaluateForX: Method*

```
#import "Equation.h"

@implementation Equation

@synthesize text;

- (id)initWithString:(NSString*)string
{
    self = [super init];
    if (self) {
        self.text = string;
    }
    return self;
}

- (float)evaluateForX:(float)x
{
    NSTask *task = [[NSTask alloc] init];
    [task setLaunchPath: @"/usr/bin/awk"];

    NSArray *arguments = [NSArray arrayWithObjects: [NSString stringWithFormat:@"BEGIN {
x=%f ; print %@ ; }", x, self.text], nil];
    [task setArguments:arguments];

    NSPipe *pipe = [NSPipe pipe];
    [task setStandardOutput:pipe];

    NSFileHandle *file = [pipe fileHandleForReading];

    [task launch];

    NSData *data = [file readDataToEndOfFile];

    NSString *string = [[NSString alloc] initWithData:data encoding:
   :NSUTF8StringEncoding];
    float value = [string floatValue];
```

```

    return value;
}

- (NSString*)description
{
    return [NSString stringWithFormat:@"Equation [%@]", self.text];
}

@end

```

We pass the `evaluateForX:` method a float value for `x`, and it uses `awk` to calculate the value for `y` and return it as a float.

Communication Among Controllers

The Graphique application has three controllers at play: `EquationEntryViewController`, `GraphTableViewController`, and `RecentlyUsedEquationsViewController`. These three controllers must be able to exchange information to tie the application together. We tie them together through the application delegate. Because all three controllers are available as properties of the application delegate, any class that can get a hold of the application delegate can get a handle to any of the controllers. This setup is handy to facilitate the communication between controllers. It is the responsibility of `GraphTableViewController` to plot curves, for example, so once we have a new equation in our `EquationEntryViewController` instance, we want to tell our `GraphTableViewController` instance to draw it.

Add a new method called `draw:` to `GraphTableViewController`. For now, the method in `GraphTableViewController.m` should look this:

```

- (void)draw:(Equation *)equation
{
    NSLog(@"Draw equation: %@", equation);
    NSLog(@"value for x=4, y=%f", [equation evaluateForX:4.0]);
}

```

Later in this chapter, we'll substitute a real implementation. Make sure to also define this method in `GraphTableViewController.h`, as shown in Listing 3–10.

Listing 3–10. *GraphTableViewController.m Declaring the New Draw Method*

```

#import <Cocoa/Cocoa.h>
#import "Equation.h"

@interface GraphTableViewController : NSViewController

- (void)draw:(Equation *)equation;

@end

```

Declare a new `IBOutlet` property of type `NSTextField` in `EquationEntryViewController.h`. Call it `textField` and link it to the equation entry text field in Interface Builder as you learned how to do in the previous chapter. See Listing 3–11.

Listing 3–11. Adding an outlet to EquationEntryViewController.h

```
#import <Cocoa/Cocoa.h>

@interface EquationEntryViewController : NSViewController

@property (weak) IBOutlet NSTextField *textField;

- (IBAction)equationEntered:(id)sender;

@end
```

We want to call the `draw:` method from the equation entry controller, passing the Equation object created from the text in `textField`, when the Graph button is clicked. Edit the `equationEntered:` method in `EquationEntryViewController` to match Listing 3–12. Be sure to import `Equation.h`, `GraphiqueAppDelegate.h`, and `GraphTableViewController.h` for the names to resolve.

Listing 3–12. The `equationEntered:` Method Communicating with `GraphTableViewController`

```
- (IBAction)equationEntered:(id)sender
{
    GraphiqueAppDelegate *delegate = NSApplication.sharedApplication.delegate;

    Equation *equation = [[Equation alloc] initWithString:[self.textField stringValue]];
    [delegate.graphTableViewController draw:equation];
}
```

In this implementation, we simply get ahold of the application delegate and send the equation to the appropriate controller. The application delegate serves as a bridge between the controllers.

In Graphique, we now allow the user to enter an equation, and we tell the appropriate controller to draw the equation, but we don't know whether the equation we're going to draw is valid. In the next section, we validate the user input.

Validating Fields

Field validation is not necessary for your application to function, but your users will expect a polished user interface and will reward you for your efforts with App Store comments leading to extra sales. You can't have a nice slick user interface without implementing field validation. By validating the input, you limit the possibilities for things to go wrong, and therefore you improve your users' satisfaction. In this section, we show you two options for validating: after the user submits the data and while the user is entering the data.

Validating After Submitting

In this section, we implement validation of the equation entry field. The validation will be invoked as the user clicks the Graph button and will display an error message when things go wrong.

Writing a Validator

The first step of any validation process is to define the rules to validate. In our Graphique application, we use a small set of rules to keep things simple. Table 3–1 shows the possible errors and the codes we associate with them. Adding an error code with each error makes it easier for your users to report problems, especially if your application is translated into several languages.

Table 3–1. *The Graphique Error Codes*

Code	Description
100	Invalid character typed. Only x()+-*/^0123456789. are allowed.
101	Consecutive operators are not allowed.
102	Too many open parentheses.
103	Too many closed parentheses.

Since we’ve been diligent about creating a model to hold our data (in other words, the Equation class), we should put our validator code inside our model. After all, what object knows more about equations than the Equation class itself? Open Equation.h and declare the following method:

```
- (BOOL)validate:(NSError **)error;
```

We also must implement the validate: method in Equation.m. The validate: method must do three things:

1. Detect any errors
2. Assign an error code and error message to any detected errors
3. Produce an NSError object for a specified error code and error message

The third item, producing an NSError object for a specified error code and error message, represents a discrete piece of functionality and should be broken out to a separate method. We don’t want code outside the Equation class to call this code, however, so we don’t want to include this method in Equation’s public interface. To achieve both these aims, putting this functionality into a method but hiding the method from outside the Equation class, we create a private category.

You can read more about categories in the Apple documentation at <http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Chapters/ocCategories.html>, but categories can be simply summed up as collections of methods that are added to a class at runtime. You could, for example, create a category that adds a method called `romanNumeralValue:` to `NSNumber` so that you can write code like this:

```
NSNumber *num = [NSNumber numberWithInt:2011];
NSLog(@"%@", [num romanNumeralValue]);
```

to get output like this:

```
2011-05-10 06:40:52.690 MMXI
```

In our case, we create a category containing a single method, `produceError:withCode:andMessage:`, and we declare the category not in a separate header file but at the top of `Equation.m` above this line:

```
@implementation Equation
```

This makes the method visible only for the `Equation` class, and we can call it just as if we'd declared the method normally in `Equation.h`. Open `Equation.m` and add the code shown in Listing 3–13. Note that we haven't shown the existing methods to save paper; don't remove them!

Listing 3–13. *The Equation Validation Method and the Private Category*

```
@interface Equation ()
- (BOOL)produceError:(NSError**)error withCode:(NSInteger)code
andMessage:(NSString*)message;
@end

@implementation Equation

// The rest of the implementation code . . .

- (BOOL)validate:(NSError **)error
{
    // Validation rules
    // 1. Only digits, letters '.', 'x', '(', ')', '+', '-', '*', '/', '^', ' ' allowed
    // 2. There should be the same amount of closing and opening parentheses
    // 3. no two consecutive operators

    // Counters for '(' and ')'
    NSUInteger open = 0;
    NSUInteger close = 0;

    NSString *allowedCharacters = @"x()+-/*^0123456789. ";

    NSCharacterSet *cs = [NSCharacterSet
characterSetWithCharactersInString:allowedCharacters];
    NSCharacterSet *operators = [NSCharacterSet characterSetWithCharactersInString:@"+*/^"];

    unichar previous = 0;

    for(NSUInteger i=0; i<text.length; i++)
    {
        unichar c = [text characterAtIndex:i];
        if(![cs characterIsMember:c])
        {
            // Invalid character

```



```

        return [self produceError:error withCode:100 andMessage:[NSString
stringWithFormat:@"Invalid character typed. Only '%@' are allowed", allowedCharacters]];
    }
    else if(c == '(') open++;
    else if(c == ')') close++;

    if([operators characterIsMember:c] && [operators characterIsMember:previous])
    {
        // Two consecutive operators
        return [self produceError:error withCode:101 andMessage:@"Consecutive operators
are not allowed"];
    }

    if(c != ' ') previous = c;
}

if(open < close)
{
    // Invalid character
    return [self produceError:error withCode:102 andMessage:@"Too many closed
parentheses"];
}
else if(open > close)
{
    // Invalid character
    return [self produceError:error withCode:103 andMessage:@"Too many open
parentheses"];
}

return YES;
}

- (BOOL)produceError:(NSError**)error withCode:(NSInteger)code
andMessage:(NSString*)message
{
    if (error != NULL)
    {
        NSMutableDictionary *errorDetail = [NSMutableDictionary dictionary];
        [errorDetail setValue:message forKey:NSLocalizedStringKey];
        *error = [NSError errorWithDomain:@"Graphique" code:code userInfo:errorDetail];
    }
    return NO;
}

```

The implementation checks for the four rules we defined earlier in this section and produces the appropriate error with the appropriate error code when necessary.

Unit Testing

Unit testing refers to the practice of writing nonproduction code that exercises a portion of the application in order to validate that it functions as expected in an isolated environment. Different developers adopt unit testing to various degrees. Some

developers insist on subjecting every single line of code to unit testing. Others are stubbornly resistant to writing any kind of unit testing code. We leave it to you to find the ideal amount of unit testing for your projects. Typically, however, algorithms are ideal candidates for unit tests. This is certainly true for our validation method.

When we created the Graphique project, Xcode generated a GraphiqueTests class in the GraphiqueTests folder. This is the main unit test harness for the application. Open GraphiqueTests.m and add one method for each test you want to run. Since we have four validation rules, we need to have at least five methods (although we encourage you to write more to test more cases). Our five cases represent a successful case and one failure case for each validation rule. For those failure cases, we check that we actually got a failure and that the failure is reporting the proper error code.

Remember our validation rules from Table 3–1. It's always best to not look at the code you're testing when writing the unit tests. It keeps them independent and unbiased.

Listing 3–14 shows how the methods are implemented. For every case where something we expected didn't happen, we invoke the STFail() function to report the unit test failure.

Listing 3–14. *GraphiqueTests.m: Unit Tests for the Equation Validation Method*

```
#import "GraphiqueTests.h"
#import "Equation.h"

@implementation GraphiqueTests

- (void)setUp
{
    [super setUp];
}

- (void)tearDown
{
    [super tearDown];
}

- (void)testEquationValidation
{
    NSError *error = nil;
    Equation *equation = [[Equation alloc] initWithString:@"( 3+4*7 /(3+ 4))"];
    if(![equation validate:&error])
    {
        STFail(@"Equation should have been valid");
    }
}

- (void)testEquationValidationWithInvalidCharacters
{
    NSError *error = nil;
    Equation *equation = [[Equation alloc] initWithString:@"invalid characters"];
    if([equation validate:&error])
    {
        STFail(@"Equation should not have been valid");
    }
}
```

```

    }

    if([error code] != 100)
    {
        STFail(@"Validation should have failed with code 100 instead of %d", [error code]);
    }
}

- (void)testEquationValidationWithConsecutiveOperators
{
    NSError *error = nil;
    Equation *equation = [[Equation alloc] initWithString:@"2++3"];
    if([equation validate:&error])
    {
        STFail(@"Equation should not have been valid");
    }

    if([error code] != 101)
    {
        STFail(@"Validation should have failed with code 101 instead of %d", [error code]);
    }
}

- (void)testEquationValidationWithTooManyOpenBrackets
{
    NSError *error = nil;
    Equation *equation = [[Equation alloc] initWithString:@"((4+3)"];
    if([equation validate:&error])
    {
        STFail(@"Equation should not have been valid");
    }

    if([error code] != 102)
    {
        STFail(@"Validation should have failed with code 102 instead of %d", [error code]);
    }
}

- (void)testEquationValidationWithTooManyCloseBrackets
{
    NSError *error = nil;
    Equation *equation = [[Equation alloc] initWithString:@"(4+3))"];
    if([equation validate:&error])
    {
        STFail(@"Equation should not have been valid");
    }

    if([error code] != 103)
    {
        STFail(@"Validation should have failed with code 103 instead of %d", [error code]);
    }
}

@end

```

NOTE: The `STFail()` function is used to report unit test failures, which is different from reporting validation failures. In the case of testing validation failures, we call `STFail()` when the code did not return an error as expected.

To launch the unit tests, you can use the shortcut `⌘+U`. Alternatively, you can click and hold the Run button on the top-left corner of the Xcode window and select Build for Testing (it may say “Test” instead, depending on the build status) from the drop-down to start running the tests, as shown in Figure 3–14.

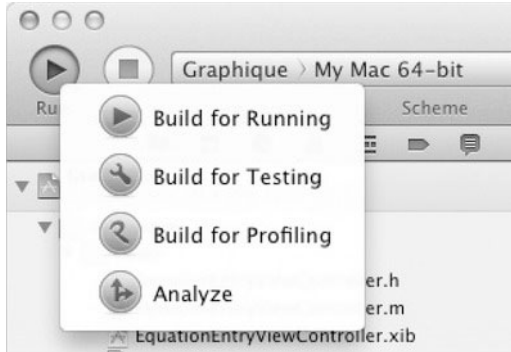


Figure 3–14. *The unit test launcher*

When you run the test, you will get some failures, as illustrated in Figure 3–15, indicating something has gone wrong.

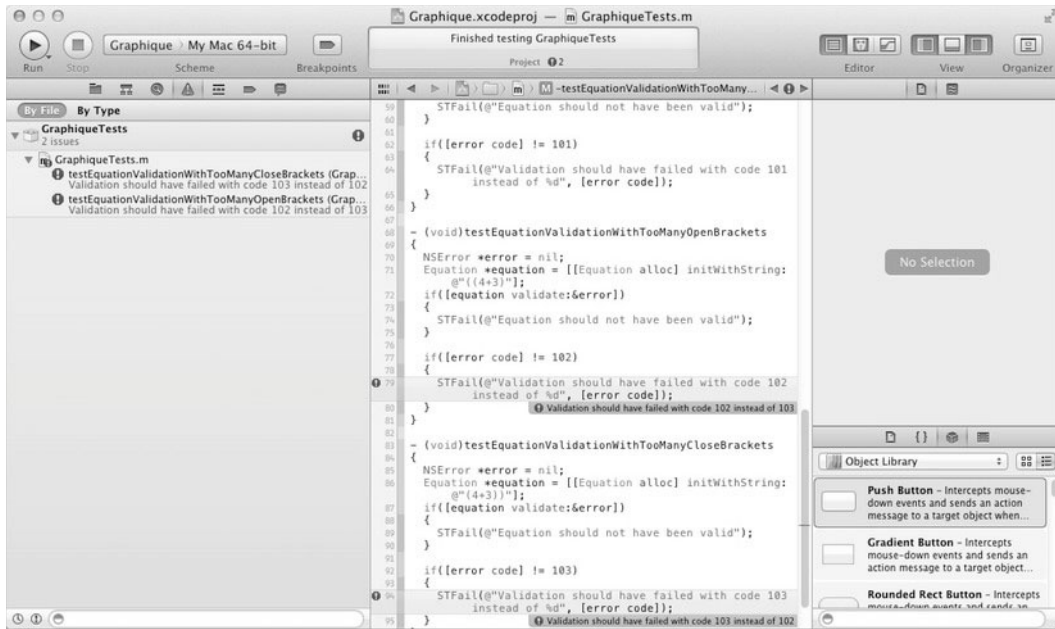


Figure 3–15. Unit test failures

The two errors are related to error codes 102 and 103 not being generated correctly. If you go back to `Equation.m`, you can find the logic for generating these two errors:

```
if(open < close)
{
    // Invalid character
    return [self produceError:error withCode:102 andMessage:@"Too many closed parentheses"];
}
else if(open > close)
{
    // Invalid character
    return [self produceError:error withCode:103 andMessage:@"Too many open parentheses"];
}
```

Whoops, we've swapped error code 102 (too many open brackets) and error code 103 (too many closed brackets). We apologize to those readers who did notice the problems before running the unit tests and thought we had made a mistake. As a reward for continuing to read this chapter, we clear up this confusion for you. Change the logic to return the right code, as shown here:

```
if(open < close)
{
    // Invalid character
    return [self produceError:error withCode:103 andMessage:@"Too many closed parentheses"];
}
else if(open > close)
{
    // Invalid character
```

```
    return [self produceError:error withCode:102 andMessage:@"Too many open parentheses"];
}
```

Now run the unit tests again. This time they should be successful, and the Console output should show something similar to the following example:

Executed 5 tests, with 0 failures (0 unexpected) in 0.000 (0.002) seconds

Displaying an Alert Window

So far, we've written our validation code and verified that it measures up to our expectations by using unit tests. It's now time to hook the validation code up to the user interface and display the error message when a problem occurs. Since we want validation performed when the user clicks the Graph button, we add our invocation to the validator in the `equationEntered:` method of the `EquationEntryViewController` class. When validation fails, we want to display a message box with the error code and message. Cocoa offers a convenient way to display message boxes using the `UIAlertView` class.

Edit the `equationEntered:` method in `EquationEntryViewController.m`. For the alert delegate to work, you must add the `alertViewDidEnd:returnCode:contextInfo:` method. It is called when the user closes the `UIAlertView` box. In our case, we don't want to do anything. It may be used to clear the input field. The code for both methods is shown in Listing 3–15.

Listing 3–15. *The Code for Displaying Alerts*

```
- (void)alertViewDidEnd:(UIAlertView *)alert returnCode:(NSInteger)returnCode
    contextInfo:(void *)contextInfo
{
}

- (IBAction)equationEntered:(id)sender
{
    NSLog(@"Equation entered");
    GraphiqueAppDelegate *delegate = [UIApplication sharedApplication].delegate;

    Equation *equation = [[Equation alloc] initWithString: [self.textField stringValue]];

    NSError *error = nil;
    if (![equation validate:&error])
    {
        // Validation failed, display the error
        UIAlertView *alert = [[UIAlertView alloc] init];
        [alert addButtonWithTitle:@"OK"];
        [alert setMessageText:@"Something went wrong. "];
        [alert setInformativeText:[NSString stringWithFormat:@"Error %d: %@", [error
code],[error localizedDescription]]];
        [alert setAlertStyle:UIAlertStyleWarning];

        [alert beginSheetModalForWindow:delegate.window modalDelegate:self
didEndSelector:@selector(alertViewDidEnd:returnCode:contextInfo:) contextInfo:nil];
    }
}
```

```
else
{
    [delegate.graphTableViewController draw: equation];
}
```

Launch the Graphique application, enter an invalid equation like **1++x**, and click the Graph button. The validation error should appear as shown in Figure 3–16.

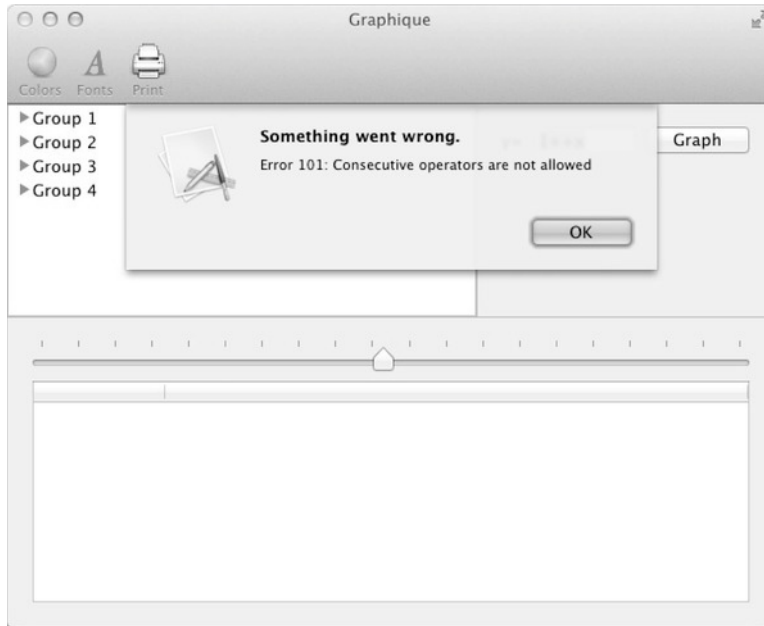


Figure 3–16. Validation error shown in an `NSAlert` message box

A Better Way: Real-Time Validation

Performing field validation is essential to providing good feedback to your users when they do something wrong, but your app will feel a lot more natural to the users if you can catch errors earlier and provide the feedback while they are doing something wrong. This section is dedicated to do just that.

Adding a Feedback Label

If we pop up an alert box every time the user types something wrong, this will undoubtedly generate a lot of frustration; users get scolded after they've done something wrong and have to then dismiss the alert. Instead, we can display the error message while they're typing in invalid equations by adding a label to the equation entry component that can hold the error message that we're currently displaying in an alert. This approach gives users real-time feedback on how they're misstepping in an unobtrusive way that doesn't require that they click an OK button. Open

EquationEntryViewController.h and declare a new NSTextField property called feedback, as Listing 3–16 shows.

Listing 3–16. *EquationEntryViewController.h with the New Feedback Text Field*

```
#import <Cocoa/Cocoa.h>
#import "Equation.h"

@interface EquationEntryViewController : NSViewController

@property (weak) IBOutlet NSTextField *textField;
@property (weak) IBOutlet NSTextField *feedback;

- (IBAction)equationEntered:(id)sender;

@end
```

Edit EquationEntryViewController.m to synthesize the new property as well.

Catching Text Change Notifications

The next step is to catch notifications sent when the equation is typed so we can validate the equation. NSTextField is a subclass of NSControl, which automatically fires a notification each time its text changes that its delegate can catch by implementing the controlTextDidChange: method.

In EquationEntryViewController.m, simply implement the method as shown in Listing 3–17. This code creates an Equation object from the text in the text field and calls its validate: method. If the validate: method returns an error, the code puts the error message in the label you created for this purpose (note that you’ve created it in code only at this point; we’ll add it to the user interface shortly). Otherwise, it clears the label.

Listing 3–17. *The controlTextDidChange: Implementation*

```
- (void)controlTextDidChange:(NSNotification *)notification
{
    Equation *equation = [[Equation alloc] initWithString: [self.textField stringValue]];

    NSError *error = nil;
    if(![equation validate:&error])
    {
        // Validation failed, display the error
        [feedback setStringValue:[NSString stringWithFormat:@"Error %d: %@", [error
code],[error localizedDescription]]];
    }
    else
    {
        [feedback setStringValue:@""];
    }
}
```

Table 3–2 shows the methods you can implement to catch text-related notifications on any subclass of NSControl.

Table 3–2. The Text-Related Notification Methods on NSControl

Method	Description
<code>-(void)controlTextDidBeginEditing:(NSNotification *)obj;</code>	Called whenever the control is about to switch to edit mode
<code>-(void)controlTextDidEndEditing:(NSNotification *)obj;</code>	Called whenever the control is about to switch out of edit mode
<code>-(void)controlTextDidChange:(NSNotification *)obj;</code>	Called whenever the control’s text has changed

Wiring It All Together in Interface Builder

So far, we’ve declared an attribute for our feedback panel and we’ve implemented the method needed to catch the change notifications, but we haven’t tied the feedback panel to a component on the interface, and we have not set our controller as the entry field’s delegate. Open `EquationEntryViewController.xib` to set this up.

First, resize the view to be tall enough to accommodate the feedback. Then, add a Wrapping Label object to the equation entry component, as shown in Figure 3–17. This will be our feedback label. Remove the title text and change the placeholder text to “Enter a function of x” or something helpful to the user. This placeholder text will appear whenever there is no title (that is, no validation error).

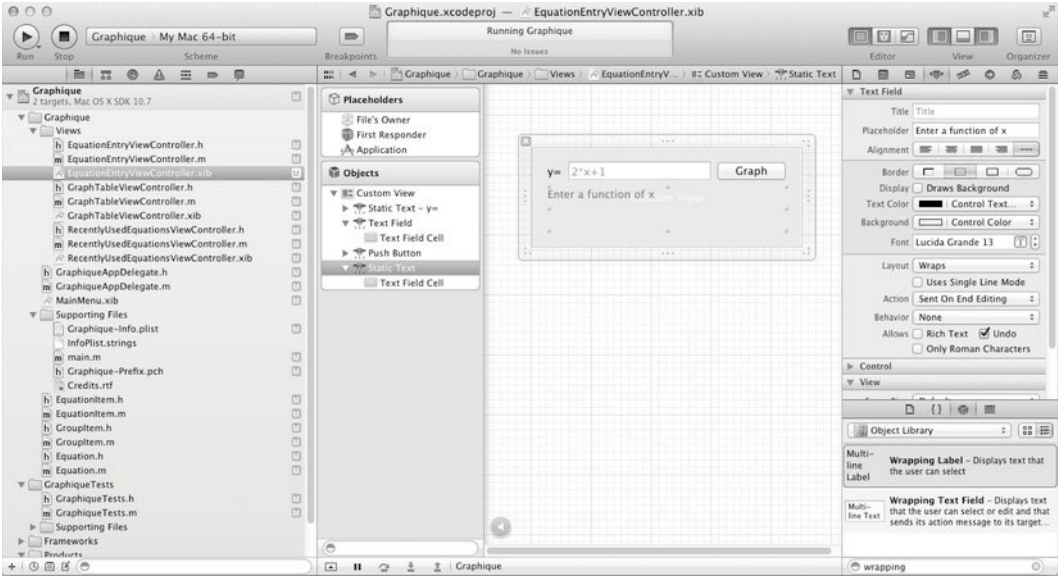


Figure 3–17. The equation entry component with a feedback label

Select the File's Owner (the controller in this case) and connect the feedback outlet to the newly added wrapping label, as illustrated in Figure 3–18.

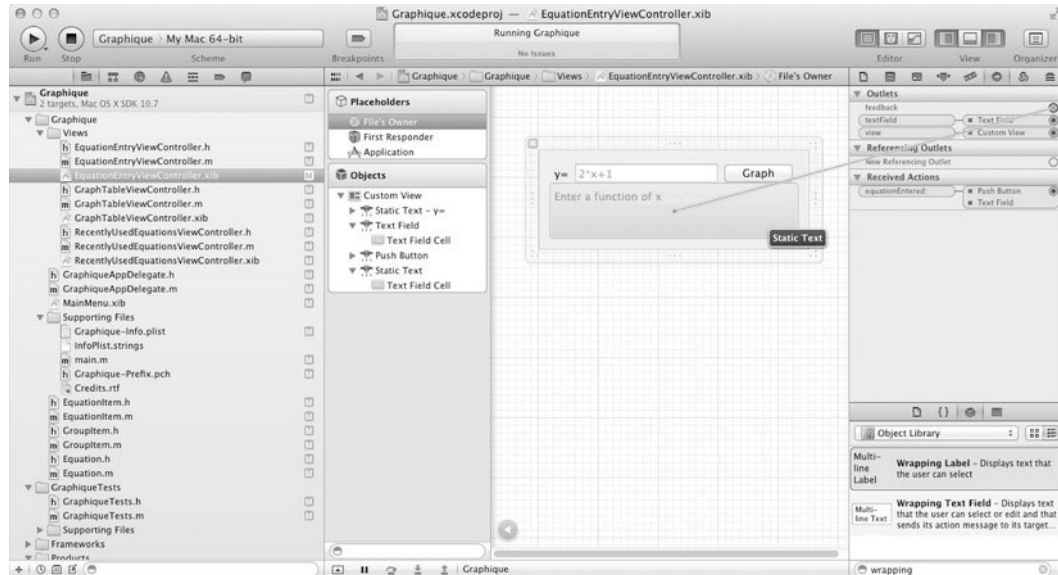


Figure 3–18. Linking the feedback attribute to the UI component

The last step is to set the controller as the delegate for the equation entry field. Select the equation entry text field and link its delegate property to the File's Owner so that it matches the illustration in Figure 3–19.

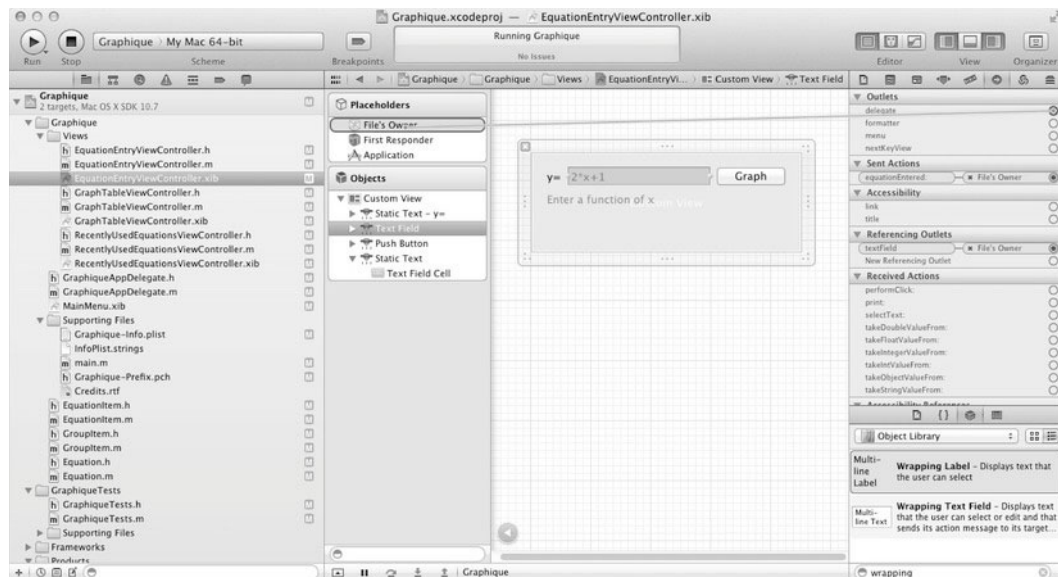


Figure 3–19. Setting the text field delegate in Interface Builder

Launch the application and type in an invalid equation to see the real-time validation in action, as shown in Figure 3–20.

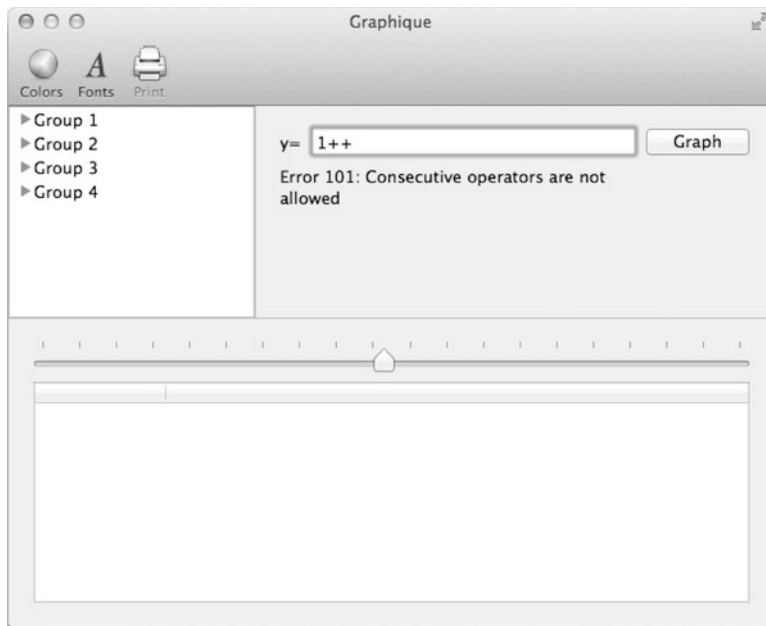


Figure 3–20. Real-time validation error

Graphing the Data

Everything we’ve done on the Graphique application to this point sets the stage for what we’re going to do in this section: graph the data. Graphing data is Graphique’s *raison d’être*, and without this capability, all the beautiful layout and responsiveness to user input means naught. In this chapter, we graph the data textually in a table with two columns: the first column, labeled X, represents the domain, and the second column, labeled Y, shows the range. This means that for a given X value, you can see its corresponding Y value in the same row in the table. We show the graph with the domain -50 to 50, with a default interval between X values of one. Later in this chapter, we allow users to select the interval between X values.

We start by putting the labels on the graph table view and setting up the alignment of the columns. In Xcode, select `GraphTableViewController.xib` and expand the Objects view, drilling down until you see the two Table Column entries below the Table View, as shown in Figure 3–21. With the first Table Column entry selected and the Attributes inspector open, enter X in the Title field and click the Center alignment button. Drag the small blue dot between the X column and second column to the right to make both columns about the same size. Switch to the Identity inspector and enter X in the Identifier field.

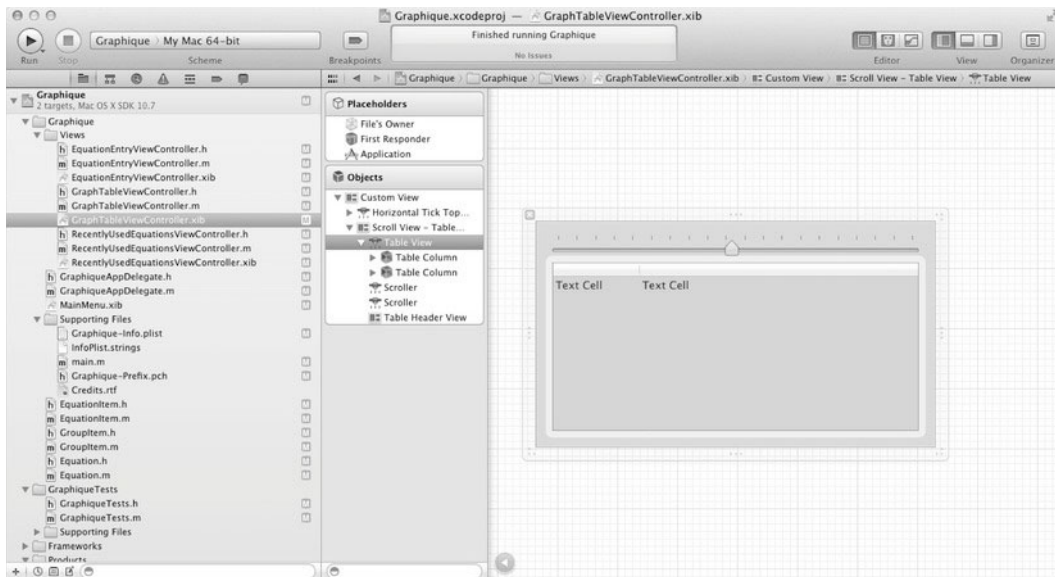


Figure 3-21. Setting up the table columns in the Graph View

Next, you want to set the data in the columns to be right-aligned, so click the triangle beside the table column you just adjusted to display the Text Field Cell item below it, and select that Text Field Cell item. In the Attributes inspector, click the Right alignment button to right-align the cell. At this point, the view should look as it does in Figure 3-22.

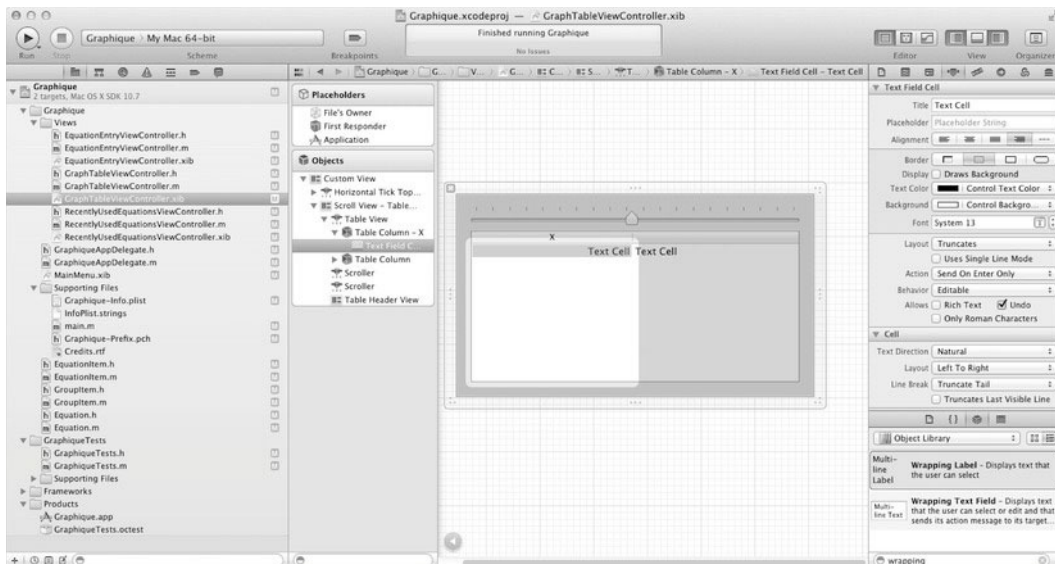


Figure 3-22. The X column set up properly

Move on to the second column and repeat the same steps, substituting Y for the label and the identifier for the column and making sure to set the alignment for the second column's Text Field Cell item as well. See Figure 3–23 for how the Graph View should now look.

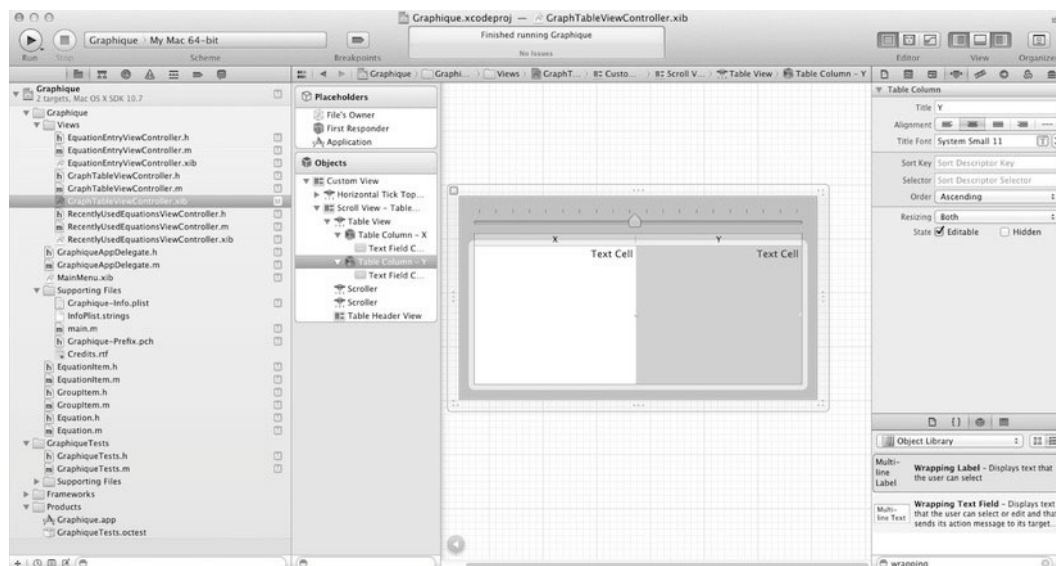


Figure 3–23. The Graph View with the columns titled X and Y

Calculating and Caching the Data

When the user clicks the Graph button, we could simply store the equation and then calculate the X and Y values for the table each time the table needs to display them (for example, as the user scrolls more rows of the table into view). This would save us the up-front time of calculating all the values for the table but might reduce the smoothness of scrolling the table if the equation is sufficiently complex.

We also could set up a cache for the data, calculating the data for a row the first time the table displays that row and caching the results so that the next time the table must display that row we can pull the information from the cache rather than recalculate it. This is often a best-of-both-worlds approach: we don't spend a lot of time up front calculating data, and the table scrolls reasonably smoothly as we fill the cache and then absolutely smoothly once we're just pulling cached values.

The third approach we could take, and in fact the one we use in Graphique, is to just calculate all the values for the table when the user clicks the Graph button and cache them. Then, as the table needs to display any of the values, we feed them directly from the cache. We take this approach for the following reasons:

- When the user clicks the Graph button, they expect something compute-intensive to happen. They can live with some amount of unresponsiveness from the application while it complies with the request to graph the data. When users scroll the table, they expect the table to just scroll and would become frustrated if the table jerked or lagged.
- We plan to graph a relatively small number of points. The domain, as we stated earlier, goes from -50 to 50, or 101 points. Later in this chapter we allow users to adjust the interval between domain points as low as 0.10, but even if they go that low, we'll be calculating a maximum of 1,010 values, not hundreds of thousands or even millions.
- In the next chapter, we display an actual graph of the equation along the same domain of -50 to 50, so we will need all the values computed so we can display the entire graph.

Most problems have many approaches, and your duty as a developer is to explore those options and arrive at a solution that best meets your users' expectations.

For Graphique, we compute all the values when the user clicks the Graph button and store each one as a `CGPoint`, which is a struct that has an `x` value and a `y` value, both floats, which is perfect for our needs. We store them in an `NSMutableArray`, taking advantage of `NSValue`'s ability to transform a `CGPoint` into an object, because `NSMutableArray` can store only objects, not primitive values like `CGPoint`. Each time the user clicks the Graph button, we clear out the cache, recompute all the values, and store them in the cache. Since we haven't yet set up the ability to talk to the table from the code, we don't yet tell the table to reload itself; we'll set that up later in the chapter.

Begin by creating an instance of `NSMutableArray` to serve as the cache. Open `GraphTableViewController.h` and add an `NSMutableArray` instance called `values`, as shown in Listing 3-18.

Listing 3-18. *GraphTableViewController.h with the Cache Added*

```
#import <Cocoa/Cocoa.h>
#import "Equation.h"

@interface GraphTableViewController : NSViewController

@property (nonatomic, retain) NSMutableArray *values;

-(void)draw:(Equation*)equation;

@end
```

Now, open `GraphTableViewController.m`, add a `@synthesize` line for the `values` member and adjust the `initWithNibName:` method to create the `values` array. Update the `draw:` method to fill the cache with the appropriate values for the equation. Note that we currently log the points to the console, because we haven't yet set up the table to display the values. See Listing 3-19.

Listing 3–19. *The Updated GraphTableViewController.m*

```
#import "GraphTableViewController.h"

@implementation GraphTableViewController

@synthesize values;

- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil {
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        self.values = [NSMutableArray array];
    }
    return self;
}

- (void)draw:(Equation*)equation
{
    // Clear the cache
    [values removeAllObjects];

    // Calculate the values
    for (float x = -50.0; x <= 50.0; x++)
    {
        float y = [equation evaluateForX:x];
        NSLog(@"Adding point (%0.2f, %0.2f)", x, y);
        [values addObject:[NSValue valueWithPoint:CGPointMake(x, y)]];
    }
}
```

Build and run Graphique, and enter **3*x+7** in the equation entry field. Click the Graph button, and inspect the Console. You should see output for all 101 points that looks like the few lines of output listed here:

```
2011-08-30 06:26:17.597 Graphique[37034:407] Adding point (-3.00, -2.00)
2011-08-30 06:26:17.603 Graphique[37034:407] Adding point (-2.00, 1.00)
2011-08-30 06:26:17.608 Graphique[37034:407] Adding point (-1.00, 4.00)
2011-08-30 06:26:17.614 Graphique[37034:407] Adding point (0.00, 7.00)
2011-08-30 06:26:17.619 Graphique[37034:407] Adding point (1.00, 10.00)
2011-08-30 06:26:17.624 Graphique[37034:407] Adding point (2.00, 13.00)
2011-08-30 06:26:17.630 Graphique[37034:407] Adding point (3.00, 16.00)
```

We are successfully calculating and caching data, but the table remains blank. Read the next section to understand how to fill the table with data.

Talking to the Table: Outlets and Delegates

You saw outlets in Chapter 2 when we created outlets for the NSSplitView instances called horizontalSplitView and verticalSplitView. As you may recall, we marked these outlets with the tag IBOutlet in the code so that Interface Builder could recognize them, and we connected the outlets in the code with actual NSSplitView instances created in Interface Builder. We're going to do the same thing with the table view in GraphTableViewController.xib, connecting it to an NSTableView instance variable in the GraphTableViewController class called graphTableView.

The other thing we're going to do to the `GraphTableViewController` class is to make it the delegate for the table view in `GraphTableViewController.xib`. Delegates are a frequently used design pattern in Cocoa, and they represent classes that conform to a known protocol that other classes can delegate functionality to. Table views use delegates for two major pieces of their functionality:

- Providing the data for the table to show (the `NSTableViewDataSource` protocol)
- Interacting with the table or customizing its view (the `NSTableViewDelegate` protocol)

Since we're just going to display data in the table for now, without providing any means for users to interact with the table (other than to scroll it), we can safely ignore the `NSTableViewDelegate` protocol, but we must implement the `NSTableViewDataSource` protocol to provide data for our `graphTableView` to display. This protocol has two methods we're going to implement: one that returns the number of rows in the table and one that returns the value for a specific cell (row and column) in the table.

Edit `GraphTableViewController.h`, declare that it implements the `NSTableViewDataSource` protocol, and add a member called `graphTableView`. The code should look like Listing 3–20.

Listing 3–20. *GraphTableViewController.h*

```
#import <Cocoa/Cocoa.h>

#import "Equation.h"

@interface GraphTableViewController : NSViewController <NSTableViewDataSource>

@property (nonatomic, retain) NSMutableArray *values;
@property (weak) IBOutlet NSTableView *graphTableView;

-(void)draw:(Equation*)equation;

@end
```

In `GraphTableViewController.m`, add a `@synthesize` line for `graphTableView`, and then open `GraphTableViewController.xib`. We're going to connect the table view instance in Interface Builder to the `graphTableView` instance in the code. Ctrl+drag from File's Owner to the table view, and in the resulting pop-up select `graphTableView`. Now, when `GraphTableViewController.xib` loads, the table view will be connected to the `graphTableView` instance.

Go back to `GraphTableViewController.m` to implement the `NSTableViewDataSource` protocol. The first method to implement returns the number of rows in the table, which should match the number of values in the values cache. The method looks like Listing 3–21.

Listing 3–21. *Method to Return the Number of Rows in the Table*

```
- (NSInteger)numberOfRowsInTableView:(NSTableView *)tableView
{
    return values.count;
}
```


The second method returns the value for a specific cell. The method is passed an `NSInteger` for the row the table needs the value for and an `NSTableColumn` instance representing the column the table is requesting the value for. Remember when we set the identifiers for the table columns to X and Y? That's what we use to determine which table column to return the value for. First, we pull the `CGPoint` instance for the row out of the cache, and then we get the value for the X or Y column, depending on which table column we've been passed. Note that we must return an object, not a primitive float. We could use `NSNumber`'s `numberWithFloat:` method to convert the float value to an `NSNumber` object and return that, but instead we return an `NSString` so we can control how the return value is formatted in the table—in this case, with two digits after the decimal point. The method implementation looks like Listing 3–22.

Listing 3–22. Method for Returning a Cell's Value

```
- (id)tableView:(NSTableView *)aTableView objectValueForTableColumn:(NSTableColumn
*)aTableColumn row:(NSInteger)rowIndex
{
    CGPoint point = [[values objectAtIndex:rowIndex] pointValue];
    float value = [[aTableColumn identifier] isEqualToString:@"X"] ? point.x : point.y;
    return [NSString stringWithFormat:@"%0.2f", value];
}
```

The last thing we must do to display data in the table is to update the `draw:` method to tell the table to reload after we've loaded the cache. We can also remove the log message that lists all the points, since we can now see the points in the table. The method now looks like Listing 3–23.

Listing 3–23. The Updated `draw:` Method

```
- (void)draw:(Equation*)equation
{
    // Clear the cache
    [values removeAllObjects];

    // Calculate the values
    for (float x = -50.0; x <= 50.0; x++)
    {
        float y = [equation evaluateForX:x];
        [values addObject:[NSValue valueWithPoint:CGPointMake(x, y)]];
    }
    [self.graphTableView reloadData];
}
```

At this point, you've updated the `GraphiqueTableViewController` class to be an adequate data source for `graphTableView`, but you haven't yet told `graphTableView` to use the `GraphiqueTableViewController` as its delegate. Open `GraphTableViewController.xib`, select the Table View, and open the Connections inspector. Drag from the circle to the right of `dataSource` to File's Owner to make that connection, as shown in Figure 3–24.

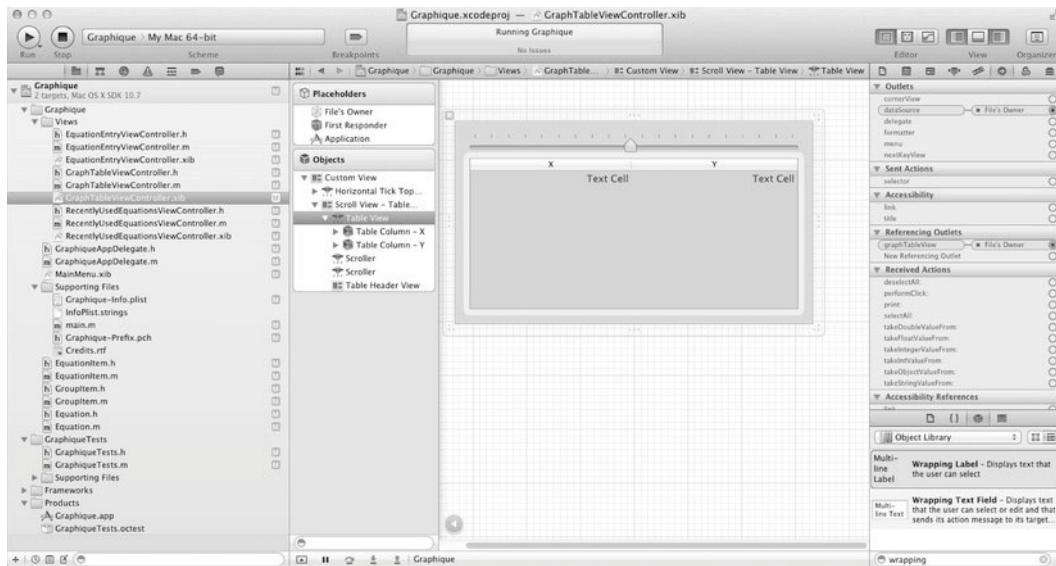


Figure 3–24. Connecting the data source for the table view

Build and run Graphique, enter $3x+7$ in the equation entry field, and click the Graph button. You should see your table fill with data, as shown in Figure 3–25. Enter various other equations in the equation entry field and click the Graph button after each one to see the table update with the new data.

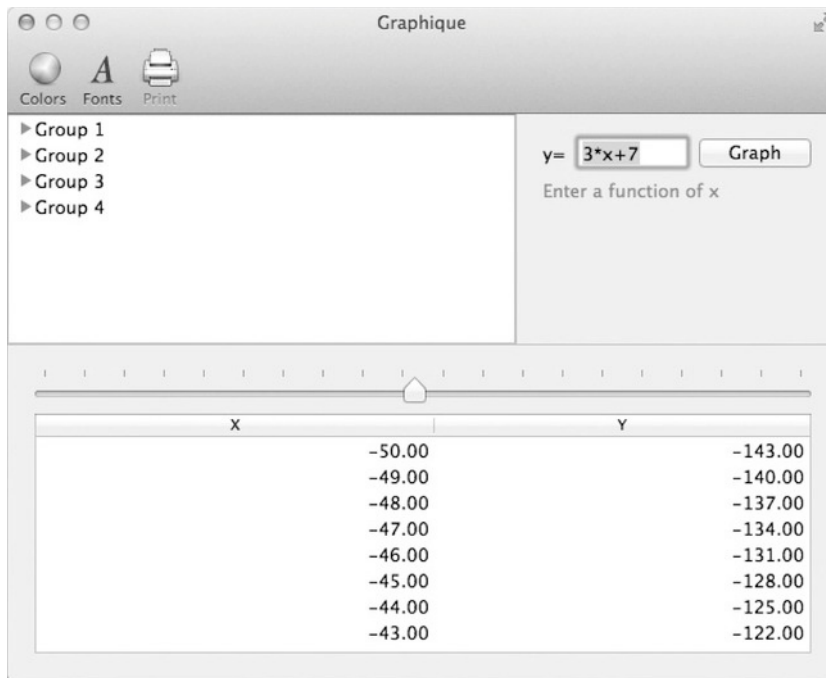


Figure 3–25. *The table with data*

Graphique now successfully graphs data by displaying values in a table. Before we close this chapter, however, we add one more ability to the application: the ability to change the interval between X values. This is the subject of the next section.

Changing the Interval in the Domain

When we laid out the Graph Table View panel, we included both a table, which we just finished filling with data, and a Slider control that we said would allow users to change the interval between X values. We now fulfill our promise to implement that feature.

When we configured that slider, we set its range in the Attributes inspector in Interface Builder from 0.10 to 5.00. As the user slides the handle left and right and clicks the Graph button, we should update the interval between X values in the table. This will mean clearing the cache and recalculating the values for the table to show. We already have a method for clearing the cache, recalculating the values, and reloading the table data called `draw:`. We must now make two changes to the Graphique application:

- Default the slider's initial position to 1
- Update the calculation loop in the `draw:` method to step by the selected interval instead of just the default of 1

To accomplish both these items, we take advantage of a Cocoa mechanism called key-value coding.

Using Key-Value Coding

Cocoa provides a mechanism called *key-value coding* (KVC) that allows you to set and get the value of a variable by its name, which we refer to as a *key*. Think of key-value coding as one entry in a map, hash, or dictionary. Suppose, for example, that you had a variable called `foo`. Its key is an `NSString` instance: `@\"foo\"`, and we could set its value using code like this:

```
[self setValue:@\"bar\" forKey:@\"foo\"];
```

This would set the value of the variable `foo` to `@\"bar\"`. We could get the value back out of `foo` by using this code:

```
[self valueForKey:@\"foo\"];
```

The value of this approach may not be immediately apparent, and we know all you compile-time safety advocates are squirming right now because compilers won't ensure that you spelled the name of the key correctly, but key-value coding can be a valuable approach to getting and setting values. You could, for example, use the same code to loop through a list of keys to set values on variables. The direct application for key-value coding for what we're doing, though, is what Cocoa calls *binding*. With binding, we can tie a graphical control—in our case, the slider—to a key, so that as we update the control, we update the value. Conversely, as we update the value, we update the control. Let's see how this works.

Open `GraphTableViewController.h` and add a `CGFloat` member called `interval`. Your file should match the code in Listing 3–24. Note that key-value coding works with objects, and `CGFloat` is just a primitive float, but Cocoa will take care of wrapping `interval` with the `NSNumber` object wrapper as necessary for getting and setting its value through key-value coding.

Listing 3–24. *GraphTableViewController.h*

```
#import <Cocoa/Cocoa.h>
#import \"Equation.h\"

@interface GraphTableViewController : NSViewController <NSTableViewDataSource>

@property (nonatomic, retain) NSMutableArray *values;
@property (weak) IBOutlet NSTableView *graphTableView;
@property (nonatomic, assign) CGFloat interval;

-(void)draw:(Equation*)equation;

@end
```

In `GraphTableViewController.m`, add an `@synthesize` line for `interval`, and in the `initWithNibName:` method, set its initial value to 1.0, as in Listing 3–25.

Listing 3–25. *The initWithNibName: Method Updated to Set interval to 1*

```
-(id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
```

```
if (self) {  
    self.values = [NSMutableArray arrayWithCapacity:0];  
    self.interval = 1.0;  
}  
return self;  
}
```

Binding the Value to the Slider

This is where key-value coding gets fun. Open `GraphTableViewController.xib`, select the slider (make sure to select the Slider Cell object), and open the Bindings inspector. Expand the Value group (underneath the Value section), check the box by Bind to:, and select File's Owner from the drop-down, which binds the slider to the owning `GraphTableViewController` instance. Then, type **`self.interval`** in the Model Key Path field, as shown in Figure 3–26. The value of the slider is now bound to the instance variable `interval` through key-value coding, and changing one will change the other (and vice versa). To prove this, build and run the application. Before, the slider started right in the middle of its range at 2.5 (OK, not exactly the middle but awfully close). Now, it starts at 1.0, as shown in Figure 3–27.

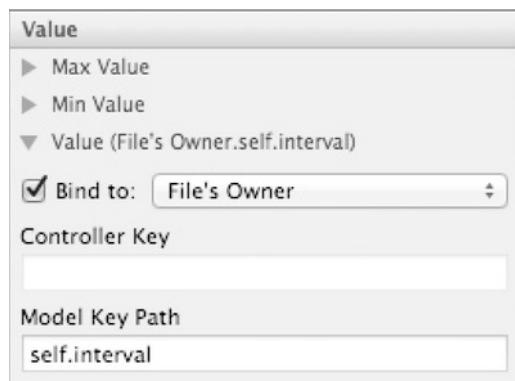


Figure 3–26. *Selecting the variable to bind the slider to*

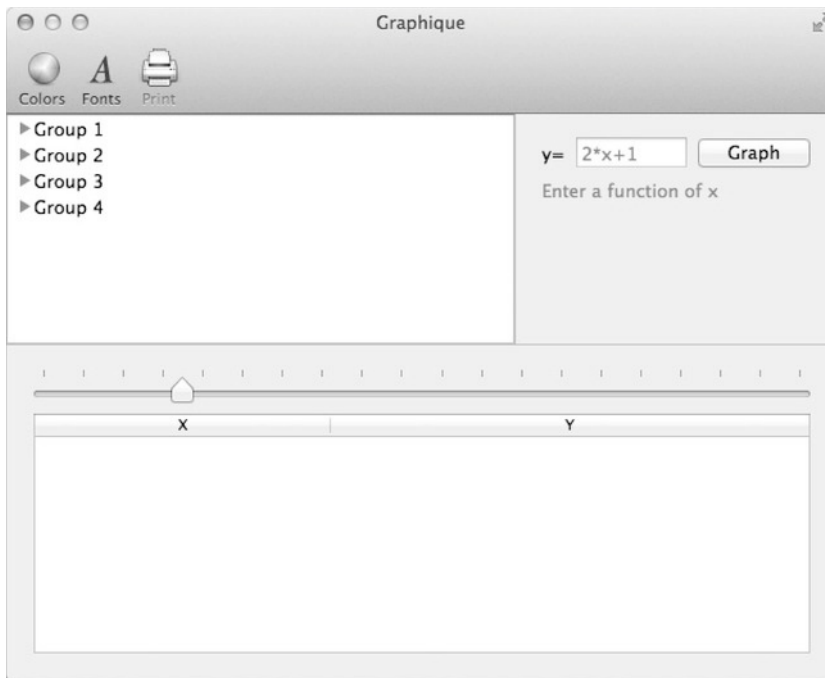


Figure 3–27. *Graphique with the slider bound to interval, set to 1.0*

You can prove this by changing the initial value you set interval to in `initWithNibName:`. Set it to 5, for example, and build and run Graphique to see that the slider starts to the far right of its range. Note that you can set interval to values beyond the slider's range without ill effects. The slider will do its best to show the value of interval by moving to its appropriate extreme, and the interval will retain the value you set. Moving the slider, however slightly, will bring interval back into its range.

The last thing we must do to incorporate the slider into the Graphique application is to step the calculation loop by the value of interval. Update the `draw:` method to match Listing 3–26.

Listing 3–26. *The draw: Method Incorporating the Interval*

```
- (void)draw:(Equation*)equation
{
    // Clear the cache
    [values removeAllObjects];

    // Calculate the values
    for (float x = -50.0; x <= 50.0; x += interval)
    {
        float y = [equation evaluateForX:x];
        [values addObject:[NSValue valueWithPoint:CGPointMake(x, y)]];
    }
    [self.graphTableView reloadData];
}
```

Build and run Graphique. Push the slider to the far right, enter x^2-3 in the equation entry field, and click Graph. You should see an interval of 5 between X values, as shown in Figure 3–28. Now, push the slider to the far left and click Graph. The interval should now be 0.1, as shown in Figure 3–29.

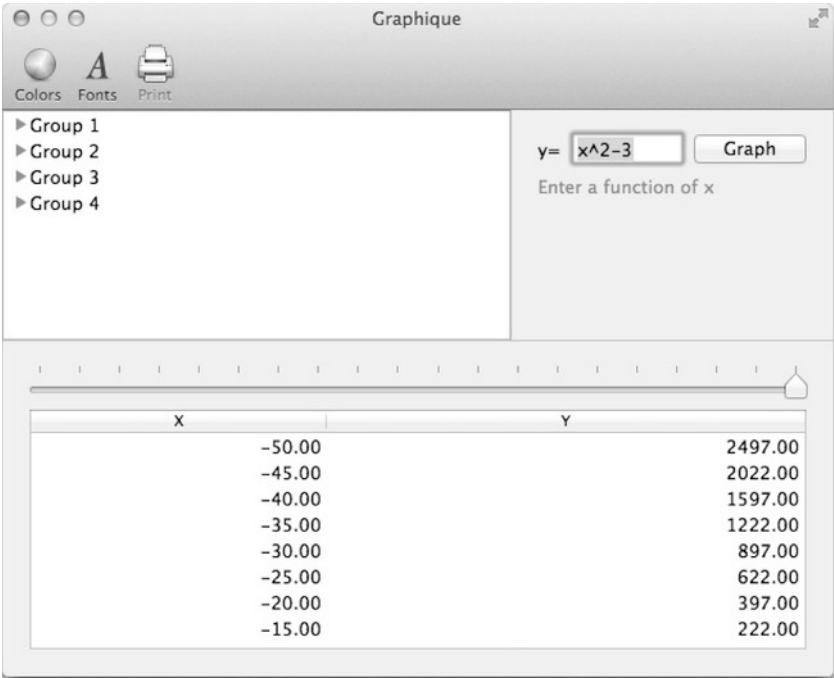


Figure 3–28. A parabola with the interval set to 5.0

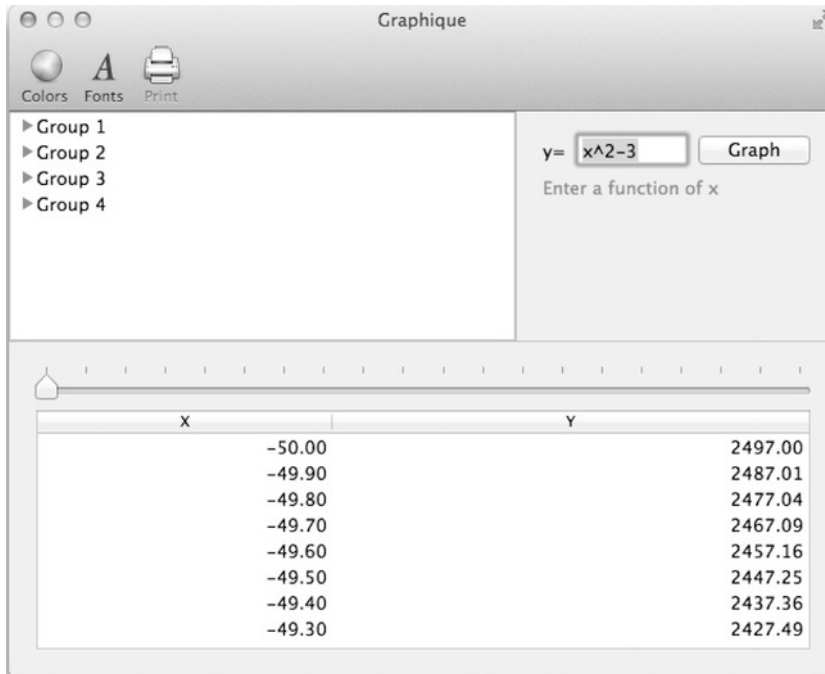


Figure 3–29. The same parabola with the interval set to 0.1

Binding the slider to the interval instance variable through key-value coding made the task of adjusting the interval simple.

Summary

If we were developing Graphique purely as an open source application, we'd call this version 0.1 and release it now. It's a working application that allows users to enter equations. It validates the equations. It displays appropriate error messages. It graphs the equations textually by showing the domain and corresponding range values. It even allows users to change the domain interval. It's ready to ship to the open source community to get feedback and invite other contributors.

The App Store community generally differs from the open source community, however. Whereas the open source community tends to be more technical and generally encourages early releases with limited functionality so that they can get involved in contributing code and shaping the direction of the application, the App Store community includes swaths of people who don't write code, don't want to mess with fledgling apps early in their development cycles, and want to use (and pay for) complete products. Graphique still lacks features important to a graphing calculator—say, for example, a graphical graph. In the next chapter, we add a graph and an improved equation entry field. Make no mistake, however: at this point, Graphique means something.

Pimp My UI

At this point in its evolution, Graphique indeed means something. It does something. It's serviceable. It accepts an equation, evaluates it, and lists x, y values. It's also a little embarrassing: it's a graphing calculator that doesn't graph. That flaw is difficult to hide or gloss over. We can't really be proud of Graphique until it graphs equations.

The equation entry field is a little shameful, too, in a Notepad kind of way: monochromatic text, stiff syntax, and anemic validation. Before we can be proud of Graphique, we must provide a better equation editor.

In this chapter, we create graphs, and we improve the equation editor. By the time we're done, the UI for Graphique will look like Figure 4–1.

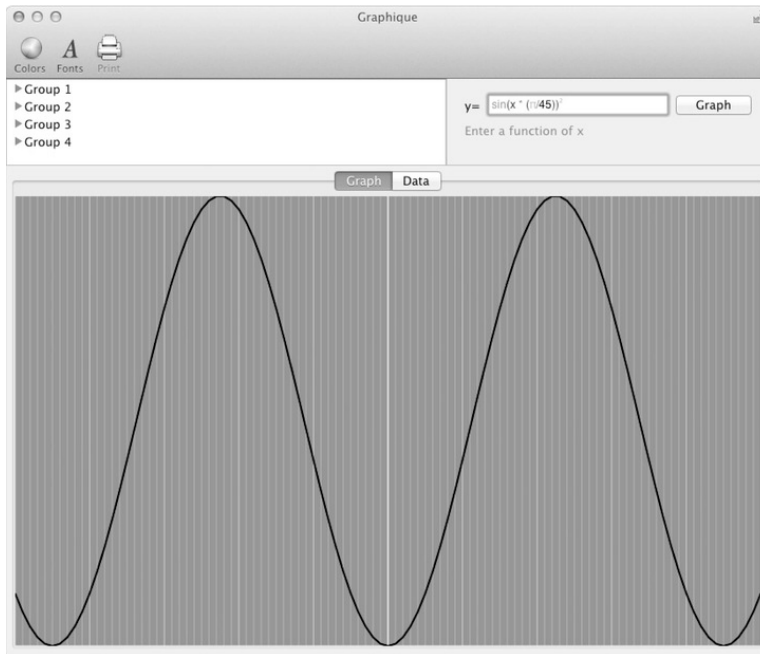


Figure 4–1. *The Graphique application at the end of this chapter*

Creating a Graph View

You've learned a lot about building user interfaces using Interface Builder (IB), but sometimes you should step off the beaten path and create user interface components that cannot be built by assembling the standard widgets that come with the platform. In this section, we go through the exercise of create a custom view that will trace our graph instead of just displaying a table of values as we've done thus far.

To create a custom view in Cocoa, you typically create a new class that subclasses the `NSView` class and implement the `drawRect:` method in order to paint your customized view on the screen.

Creating a Custom View

The first step is to create, inside the Views group, a new Cocoa Objective-C class called `GraphView` and make it a subclass of `NSView`. This will create the usual `GraphView.h` and `GraphView.m` files. Leave the implementation as is for now; we will get back to it shortly.

1. Open `GraphTableViewController.xib` and drag a new Custom View object to the Objects panel. The new custom view should be a top-level object, a sibling of the already existing Custom View object.
2. Select the new Custom View object and change the class to `GraphView` in the Identity inspector. Your setup should match Figure 4-2.

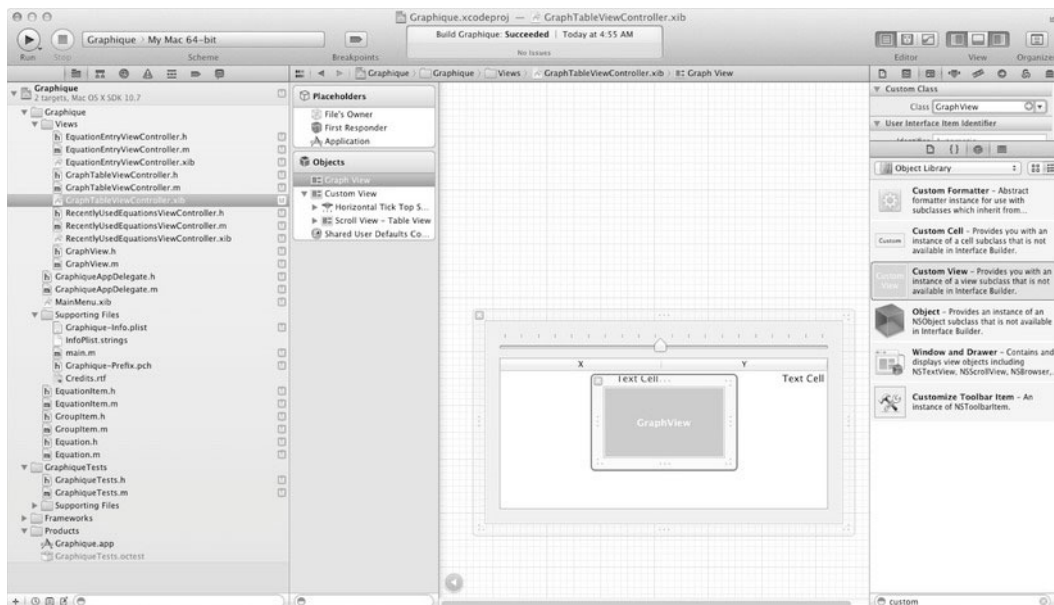


Figure 4-2. The Graph View in Interface Builder

Linking the New Custom View to the Controller

If you ran the application at this point, you would not notice anything different. This is because the `GraphTableViewController`'s view is still set to the table view. For our purpose, we change the view to point to our new `GraphView`. This will activate `GraphView` and deactivate the table view. In the next section, we show you how to keep both views active, but for now we'll leave only `GraphView` active. Select the File's Owner and go to the Connections inspector. Change the view attribute to point to Graph View, as shown in Figure 4-3.

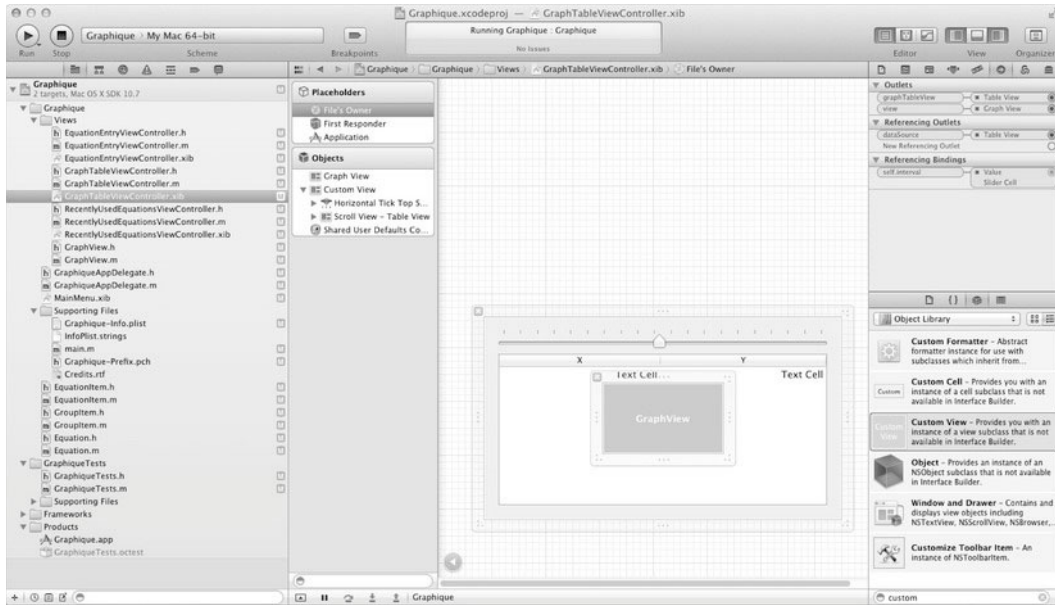


Figure 4-3. The controller pointing to a different view

You can try running the app again. This time nothing is displayed or graphed when you enter an equation. This is because you have linked the controller's view to the new `GraphView`, which doesn't yet have any code for painting anything. Before we get to the painting part, there is still a bit of wiring to do. First, we need to make sure the controller knows how to access the `GraphView`. Second, we need to make sure our `GraphView` knows how to get a hold of the data points it needs to plot.

Open `GraphTableViewController.h` and add a new `IBOutlet` property of type `GraphView`, as shown in Listing 4-1.

Listing 4-1. *GraphTableViewController.h with a Reference to GraphView*

```
#import <Cocoa/Cocoa.h>

#import "Equation.h"

@class GraphView;
```

```

@interface GraphTableViewController : NSViewController <NSTableViewDataSource>

@property (nonatomic, retain) NSMutableArray *values;
@property (weak) IBOutlet NSTableView *graphTableView;
@property (nonatomic, assign) CGFloat interval;
@property (weak) IBOutlet GraphView *graphView;

-(void)draw:(Equation*)equation;

@end

```

Be sure to synthesize the new `graphView` property in the `GraphTableViewController.m` implementation file and add the appropriate import: `#import "GraphView.h"`. Then go to `GraphTableViewController.xib` and follow the usual procedure for linking properties: select the File's Owner and open the Connections inspector. Link the `graphView` property to the Graph View object.

We now turn our attention to `GraphView`. The controller is responsible for calculating the points to plot. The view needs a way to get to where the points are stored. Open `GraphView.h` and add a new `IBOutlet` property of type `GraphTableViewController`, as shown in Listing 4–2.

Listing 4–2. *GraphView.h with a Handle to the Controller*

```

#import <Cocoa/Cocoa.h>

@class GraphTableViewController;

@interface GraphView : NSView

@property (assign) IBOutlet GraphTableViewController *controller;

@end

```

Now open `GraphView.m`, import the `GraphTableViewController.h` header file, and synthesize the property, as shown in Listing 4–3.

Listing 4–3. *GraphView.m with the Synthesized Property*

```

#import "GraphView.h"
#import "GraphTableViewController.h"

@implementation GraphView

@synthesize controller;

- (id)initWithFrame:(NSRect)frame
{
    self = [super initWithFrame:frame];
    if (self)
    {
        // Initialization code here.
    }

    return self;
}

```

```
- (void)drawRect:(CGRect)dirtyRect
{
    // Drawing code here.
}
```

```
@end
```

Go back to `GraphTableViewController.xib`. This time, select the Graph View object and go to the Connections inspector. Link the controller property to the File's Owner.

In order to make sure everything is linked properly, select File's Owner and verify that the connections match Figure 4–4.

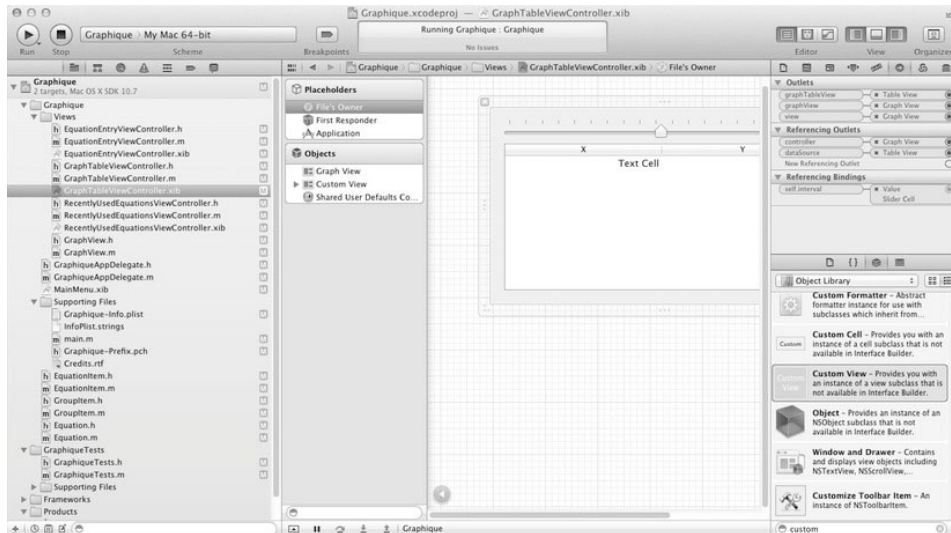


Figure 4–4. *The GraphTableViewController with all its connections*

The last step to add the custom Graph View is to tell the custom view to repaint itself whenever a new equation has been entered. For this, we add a call to `NSView`'s `setNeedsDisplay:` method in the controller, as shown in Listing 4–4.

Listing 4–4. *GraphTableViewController.m with a Call to Refresh the Custom View*

```
-(void)draw:(Equation*)equation
{
    // Clear the cache
    [values removeAllObjects];

    // Calculate the values
    for (float x = -50.0; x <= 50.0; x += interval)
    {
        float y = [equation evaluateForX:x];
        [values addObject:[NSValue valueWithPoint:NSMakeRange(x, y)]];
    }
    [self.graphTableView reloadData];

    [self.graphView setNeedsDisplay:YES];
}
```

NOTE: We've kept the call to reload the table data as well. This is because we intend to show both graph views (table and plot) later in this section, so we keep them both up-to-date.

Plotting the Graph

Now that everything is properly wired, it is time to implement the drawing function. Cocoa calls the `NSView drawRect:` method when a custom view needs to be painted. The method is given an area that it deems dirty and needs repainting. The first time the view is painted, that area is the size of the view. If the view is resized, then the `drawRect:` method is called again with the new size. We simply override this method in `GraphView` in order to take control of the painting of our custom view.

In Cocoa, most of the drawing functions are done via the `NSBezierPath` class. This class provides a way to create paths to draw from the single line to complex parametric curves. In our graphic calculator, we simply draw lines between the sampled points of the curve.

Finding the Boundaries

To allow the view to resize cleanly, we want to scale the curve to fit neatly inside the view. In order to achieve this, we must go through all the points to determine the minimum and maximum values of the curve. Once we find the range and domain extremes, we use the view's dimension, through its `bounds` property, in order to compute the scaling factors. The code to do all this, shown in Listing 4–5, goes at the top of `GraphView`'s `drawRect:` method.

Listing 4–5. *Computing the Horizontal and Vertical Scaling Factors*

```
float minDomain = CGFLOAT_MAX;
float maxDomain = CGFLOAT_MIN;

float minRange = CGFLOAT_MAX;
float maxRange = CGFLOAT_MIN;

for (NSValue *value in controller.values)
{
    NSPoint point = [value pointValue];
    if(point.x < minDomain) minDomain = point.x;
    if(point.x > maxDomain) maxDomain = point.x;

    if(point.y < minRange) minRange = point.y;
    if(point.y > maxRange) maxRange = point.y;
}

float hScale = self.bounds.size.width / (maxDomain - minDomain);
float vScale = self.bounds.size.height / (maxRange - minRange);
```

Painting the Background

The next step consists in setting up the colors we want to use and painting the background. Colors are defined by the `NSColor` class, which offers several ways of defining a color. You have options to choose predefined colors like `[NSColor whiteColor]` or use component-based colors like RGBA (Red, Green, Blue, and Alpha transparency) using `colorWithDeviceRed:green:blue:alpha:.` In our case, we use RGBA colors. Painting the rectangular background is an easy task since the `drawRect:` method gives us the rectangular area to paint and Cocoa has the `NSRectFill` function already defined. Listing 4–6 shows how to set up the colors and paint the background. It sets up colors for the background, the axes, the grids, and the curve (or actual graph).

Listing 4–6. *Setting the Colors and Painting the Background*

```
NSColor *background = [NSColor colorWithDeviceRed:0.30 green:0.58 blue:1.0 alpha:1.0];
NSColor *axisColor = [NSColor colorWithDeviceRed:1.0 green:1.0 blue:1.0 alpha:1.0];
NSColor *gridColorLight = [NSColor colorWithDeviceRed:1.0 green:1.0 blue:1.0 alpha:0.5];
NSColor *gridColorLighter = [NSColor colorWithDeviceRed:1.0 green:1.0 blue:1.0
alpha:0.25];
NSColor *curveColor = [NSColor colorWithDeviceRed:.0 green:0.0 blue:0 alpha:1.0];

[background set];
NSRectFill(dirtyRect);
```

Plotting the Graph

Now we're ready to start drawing the actual plot of our graph. In Cocoa, lines are defined using `NSBezierPath` just like any other paths. As you can probably imagine, a line (in the computer graphics sense of the term at least) is defined by a start and an end point. Mathematically oriented people would probably much rather call it a segment, but in Cocoa, it's a line. Let's take a look at what it takes to draw the domain axis.

First, we declare the new path:

```
NSBezierPath *domainAxis = [NSBezierPath bezierPath];
```

Next, we specify how thick we want the line to be. In our case, we're happy with a 1-pixel line:

```
[domainAxis setLineWidth:1];
```

Then it's just a matter of defining the beginning and the end of the line. Notice how we scale them using the scaling factors we defined earlier:

```
NSPoint startPoint = { 0, -minRange * vScale};
NSPoint endPoint = { self.bounds.size.width, -minRange * vScale };
[domainAxis moveToPoint:startPoint];
[domainAxis lineToPoint:endPoint];
```

The last step is to define the color to use and then stroke the line:

```
[axisColor set];
[domainAxis stroke];
```

We repeat the same procedure for the range axis, adding a pretty background grid and even for the curve itself, where we draw lines between each point. Listing 4–7 shows the full `drawRect:` method.

Listing 4–7. *The `GraphView`'s `drawRect:` Method to Plot the Graph*

```
- (void)drawRect:(CGRect)dirtyRect
{
    // Step 1. Find the boundaries

    float minDomain = CGFLOAT_MAX;
    float maxDomain = CGFLOAT_MIN;

    float minRange = CGFLOAT_MAX;
    float maxRange = CGFLOAT_MIN;

    for (NSValue *value in controller.values) {
        NSPoint point = [value pointValue];
        if(point.x < minDomain) minDomain = point.x;
        if(point.x > maxDomain) maxDomain = point.x;

        if(point.y < minRange) minRange = point.y;
        if(point.y > maxRange) maxRange = point.y;
    }

    float hScale = self.bounds.size.width / (maxDomain - minDomain);
    float vScale = self.bounds.size.height / (maxRange - minRange);

    // Step 2. Paint the background

    NSColor *background = [NSColor colorWithDeviceRed:0.30 green:0.58 blue:1.0
alpha:1.0];
    NSColor *axisColor = [NSColor colorWithDeviceRed:1.0 green:1.0 blue:1.0 alpha:1.0];
    NSColor *gridColorLight = [NSColor colorWithDeviceRed:1.0 green:1.0 blue:1.0
alpha:0.5];
    NSColor *gridColorLighter = [NSColor colorWithDeviceRed:1.0 green:1.0 blue:1.0
alpha:0.25];
    NSColor *curveColor = [NSColor colorWithDeviceRed:.0 green:0.0 blue:0 alpha:1.0];

    [background set];
    NSRectFill(dirtyRect);

    // Step 3. Plot the graph

    if(controller.values.count == 0) return;

    // Paint the domain axis
    {
        NSBezierPath *domainAxis = [NSBezierPath bezierPath];
        [domainAxis setLineWidth: 1];
        NSPoint startPoint = { 0, -minRange * vScale};
        NSPoint endPoint = { self.bounds.size.width, -minRange * vScale };
        [domainAxis moveToPoint:startPoint];
        [domainAxis lineToPoint:endPoint];
        [axisColor set];
        [domainAxis stroke];
    }
}
```



```

}

// Paint the range axis
{
    NSBezierPath *rangeAxis = [NSBezierPath bezierPath];
    [rangeAxis setLineWidth: 1];
    NSPoint startPoint = { -minDomain * hScale, 0 };
    NSPoint endPoint = { -minDomain * hScale, self.bounds.size.height };
    [rangeAxis moveToPoint:startPoint];
    [rangeAxis lineToPoint:endPoint];
    [axisColor set];
    [rangeAxis stroke];
}

// Paint the grid. Every 10 steps, we use a less transparent grid path for major lines
{
    NSBezierPath *grid = [NSBezierPath bezierPath];
    NSBezierPath *lighterGrid = [NSBezierPath bezierPath];

    for(int col=minDomain; col<maxDomain; col++)
    {
        NSPoint startPoint = { (col - minDomain) * hScale, 0};
        NSPoint endPoint = { (col - minDomain) * hScale, self.bounds.size.height };
        if(col % 10 == 0)
        {
            [grid moveToPoint:startPoint];
            [grid lineToPoint:endPoint];
        }
        else
        {
            [lighterGrid moveToPoint:startPoint];
            [lighterGrid lineToPoint:endPoint];
        }
    }

    int vStep = pow(10, log10(maxRange - minRange)-2);
    if(vStep == 0) vStep = 1;

    for(int row=-vStep; row>=minRange; row += vStep)
    {
        NSPoint startPoint = { 0, (row - minRange) * vScale};
        NSPoint endPoint = { self.bounds.size.width * hScale, (row - minRange) * vScale };

        if(row % (vStep*10) == 0)
        {
            [grid moveToPoint:startPoint];
            [grid lineToPoint:endPoint];
        }
        else
        {
            [lighterGrid moveToPoint:startPoint];
            [lighterGrid lineToPoint:endPoint];
        }
    }
}

```

```

for(int row=vStep; row<maxRange; row += vStep)
{
    NSPoint startPoint = { 0, (row - minRange) * vScale};
    NSPoint endPoint = { self.bounds.size.width * hScale, (row - minRange) * vScale };

    if(row % (vStep*10) == 0)
    {
        [grid moveToPoint:startPoint];
        [grid lineToPoint:endPoint];
    }
    else
    {
        [lighterGrid moveToPoint:startPoint];
        [lighterGrid lineToPoint:endPoint];
    }
}

[gridColorLighter set];
[lighterGrid stroke];
[gridColorLight set];
[grid stroke];
}

// Paint the curve
{
    NSBezierPath *curve = nil;

    for(NSValue *value in controller.values)
    {
        NSPoint point = [value pointValue];
        NSPoint pointForView = { (point.x - minDomain) * hScale, (point.y - minRange) *
vScale };

        if(curve == nil)
        {
            curve = [NSBezierPath bezierPath];
            [curve setLineWidth:2.0];
            [curve moveToPoint:pointForView];
        }
        else
        {
            [curve lineToPoint:pointForView];
        }
    }

    [curveColor set];
    [curve stroke];
}
}

```

Of course, paths don't all have to be lines. Notice in the previous listing how the curve path is defined. We put the entire path together by calling `lineToPoint:`, but the path is not actually drawn until we're out of the loop and call the `stroke` method. `NSBezierPath`

instances are constructed using lines but also using other shapes like ellipses or rectangles.

Now that the custom view is implemented, you can start the app, enter an equation, and watch it be drawn, as illustrated in Figure 4–5.

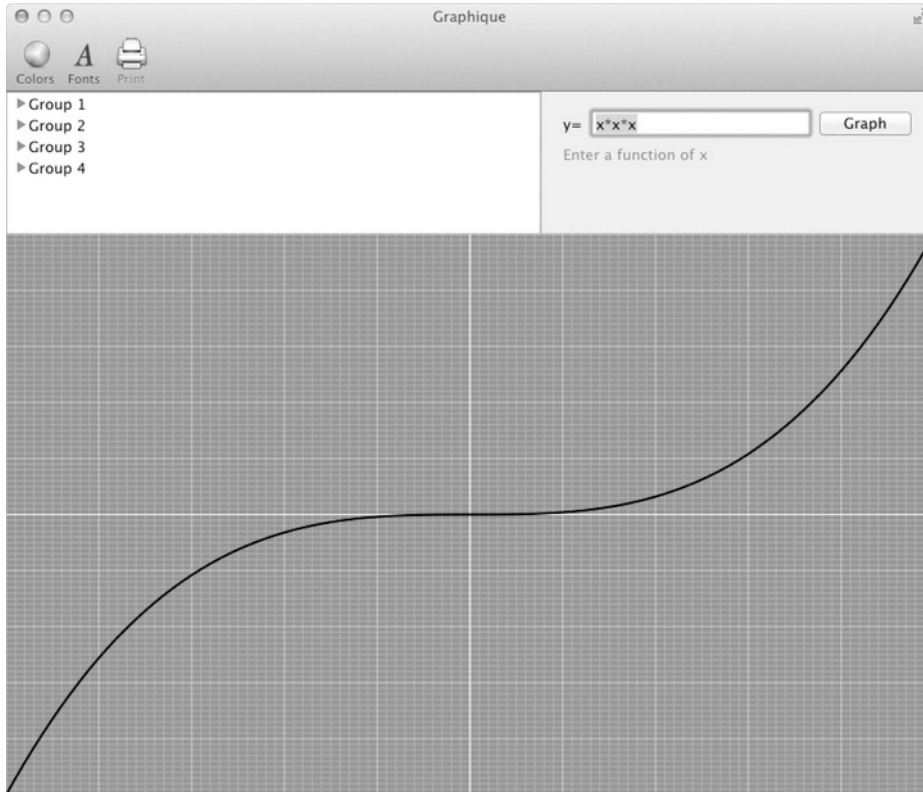


Figure 4–5. *Graphique with a plotted equation*

Toggling Between Text and Graph

So far we’ve created two representations of an equation. First we built a UI component that displays the equation data in a table. We also built another UI component that graphically plots the equation. But we haven’t been able to have both of them attached to the application at the same time so we can toggle between them. In this section, we bridge this gap by adding a tab view. Tab views are a type of view that is already outfitted with tab controls so you can navigate through subviews you attach to each tab. In this section, we replace the view of the `GraphTableViewController` with a tab view and set up two tabs, one for each representation of an equation.

Adding the Tab View

Start by opening `GraphTableViewController.xib`. We're first going to focus on making sure the anchors and resizing masks are properly set on the two existing views. Each of them should be set to resize in all directions and be anchored to all edges. This step is taken to ensure that the graph views will occupy all the space they can within the tab views, as shown in Figure 4–6.

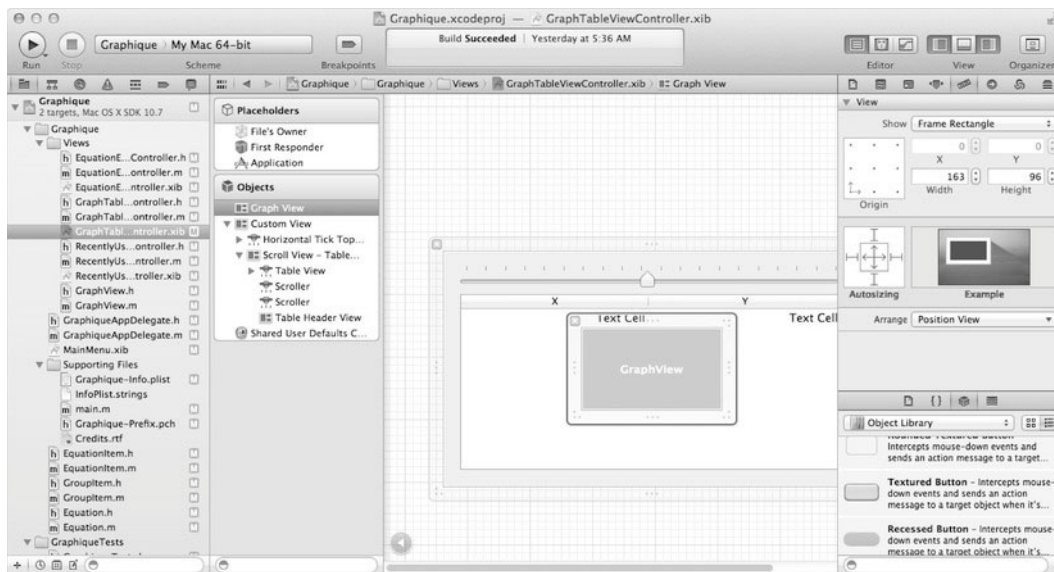


Figure 4–6. *The graph views set to maximize in all directions*

The next step is to actually add the tab view, which will eventually become the view associated with the `GraphTableViewController` controller.

Find `Tab View` in the `Object Library` and drag it to the `Objects` panel as a sibling of the existing views, as illustrated in Figure 4–7.

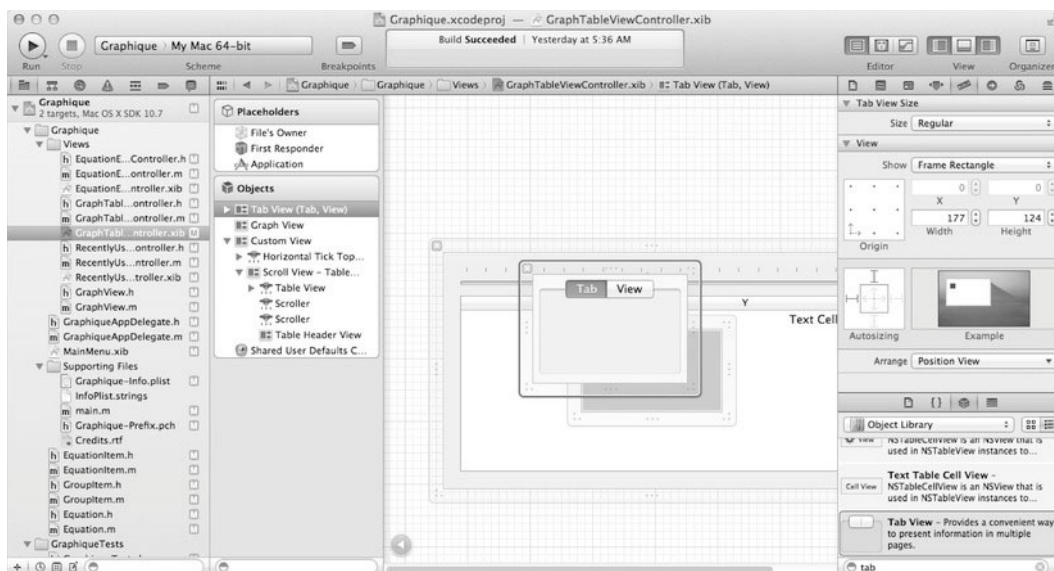


Figure 4-7. Adding the tab view

Expand the Tab View item to reveal its structure, as depicted in Figure 4-8. By default, the tab view has two tabs. You can easily add more tabs by going to the Attributes inspector and increasing the number of tabs. For our goal, two tabs works just fine.

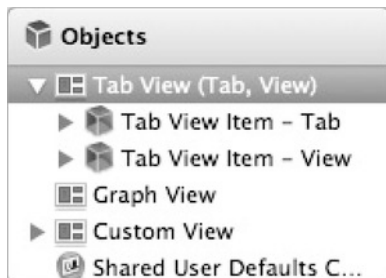


Figure 4-8. The expanded tab view structure

Adding the Views to the Tabs

We now need to place each of our equation views inside a tab. From the Objects panel, drag and drop the Graph View component onto the View component of the first tab. This will move that component inside the tab. Follow the same procedure with the Custom View component and place it inside the second tab. The resulting structure should look like Figure 4–9.

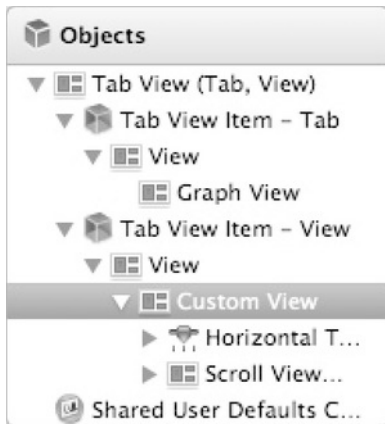


Figure 4–9. *The new tab structure with the components in place*

To make sure everything fits properly, select the Graph View object and go to the Size inspector tab. In the Arrange drop-down, select Fill Container Horizontally and Fill Container Vertically, as Figure 4–10 shows. This properly places the view in the Interface Builder designer so it looks right.

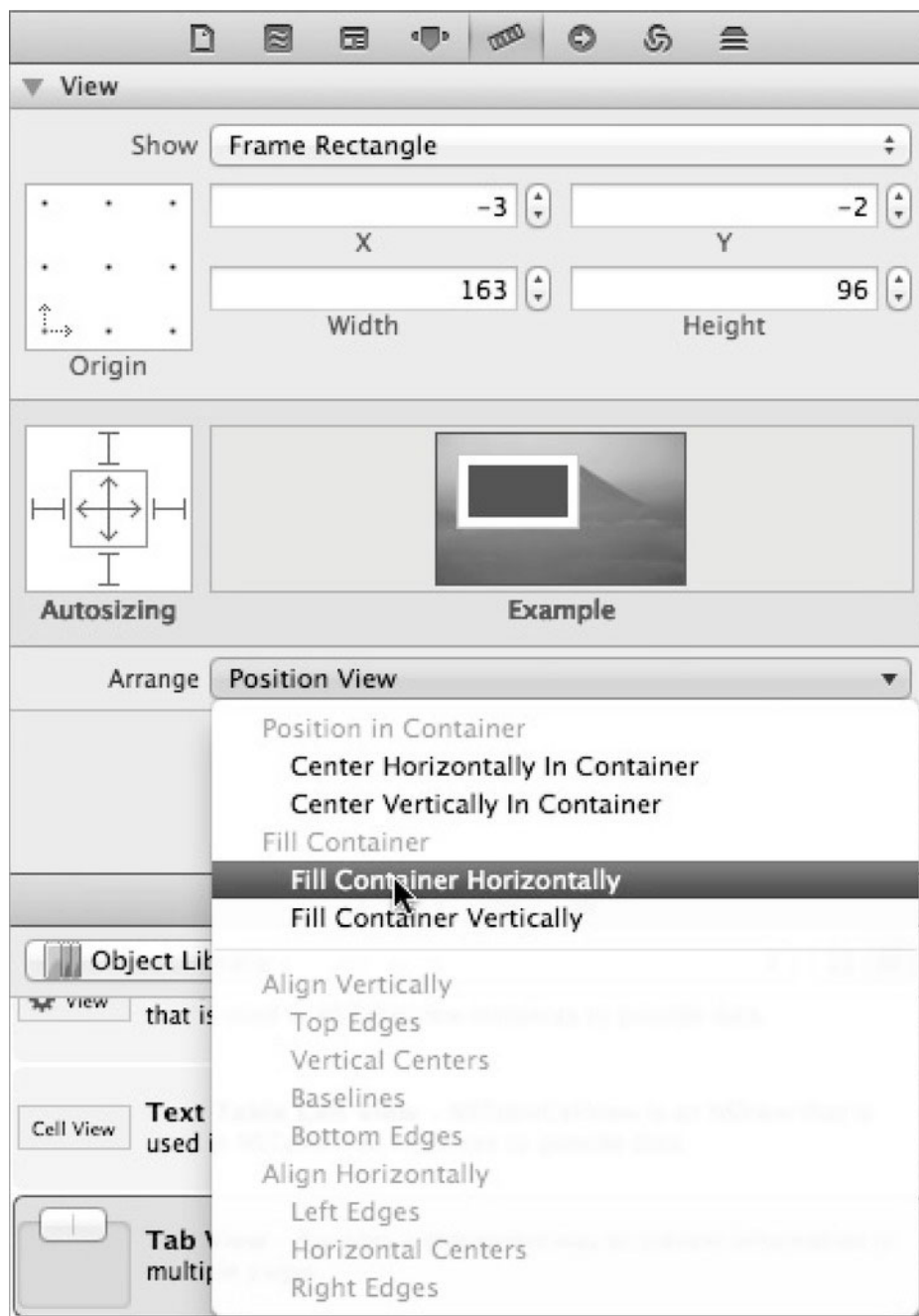


Figure 4-10. Automatically filling the container

Because the table view component has several UI components in it, it is faster to adjust them manually in the design panel. Resize the tab view to make it bigger so you see

what you are doing and move the slider and table to the right position inside the Custom View. You can also use the Arrange drop-down again and then adjust manually. The resulting view should be similar to Figure 4–11.

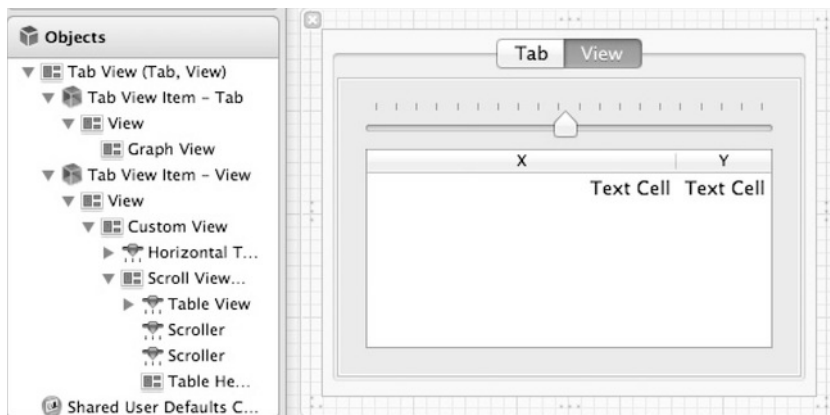


Figure 4–11. The table view properly set up inside its tab view

Each tab has a title that helps the user navigate and figure out what to expect from each tab. To change the tab's title, double-click the tab you want to rename and type the new name. We named our tabs **Graph** and **Data**.

Switching the Controller to the Tab View

Now that the individual views are in place, you want to switch the controller's view to the tab view. Select the File's Owner and open the Connections inspector. The view outlet should be attached to the Graph View. Remove that connection by clicking the x next to the connection. Once the connection is cleared, make the connection between the view outlet and the Tab View object by dragging from the empty circle to the Tab View object. The connections should be as shown in Figure 4–12.

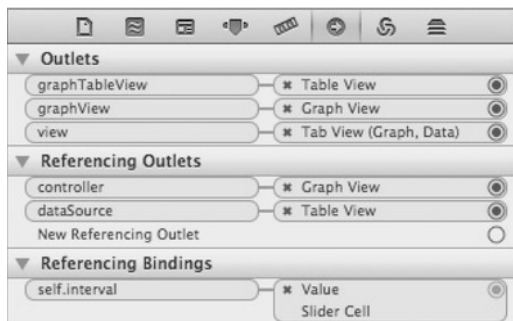


Figure 4–12. The connection from the File's Owner to the tab panel

Launch the app, enter $x*x*x$ for the equation, and click Graph. Then, select each tab, in turn, to see both the Graph View and the table view. They should match Figure 4–13 and Figure 4–14.

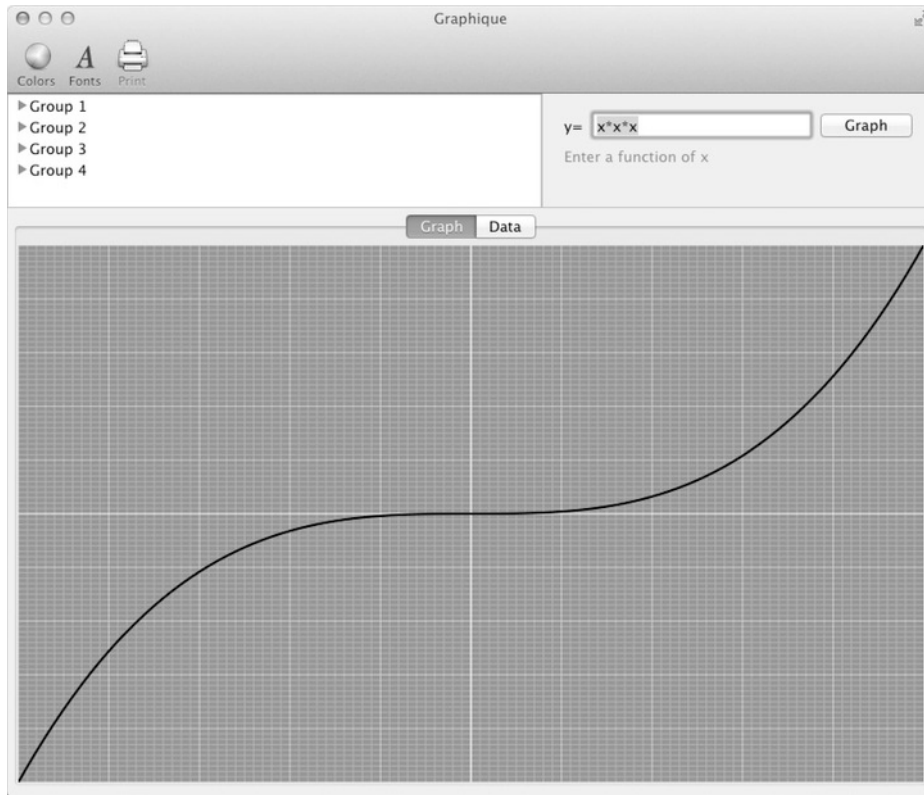


Figure 4–13. The Graphique application with tabs showing the Graph View

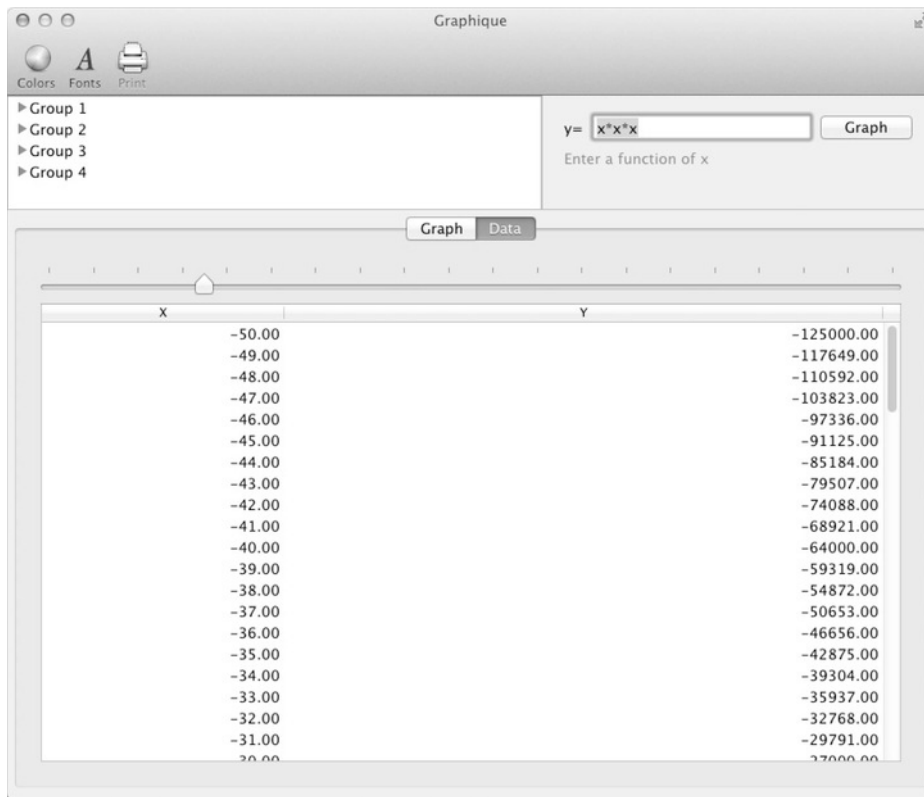


Figure 4-14. The Graphique application with tabs showing the table view

Creating a Smarter Equation Editor

We've given users a text field in which to type equations, and that text field works adequately. It's gotten us this far, after all. You wouldn't call it user-friendly or attractive, however. It's all-black text on a white background, for example, requiring users to look closely to differentiate operators from numbers. You can fool its validator by doing things like entering `)x+7(` so that the number of open parentheses match the number of closed ones, but they're out of order. Exponents stand the same height as other numbers rather than displaying in superscript. Implicit multiplication (for example, `"2x"` vs. `"2*x"`) doesn't work. We can improve this.

Over the rest of this chapter, we improve the equation entry editor by adding the following:

- Syntax coloring
- Parenthesis matching
- Inline error highlighting
- Support for trigonometric functions

- Implicit multiplication
- Superscript exponents
- Implicit exponents
- Symbols (pi and e)

These improvements will touch the equation entry field, the equation validator, and the equation evaluator, and they will require a fair amount of work to complete. Through the rest of this chapter, you'll build these improvements, and at the end of the chapter, you'll be rewarded by a better equation entry editor.

Adding Attributes to the Equation Entry Field

In most widget toolkits, text fields support a single text color, font, and text size. To get the fancy formatting that we want for our equation editor, you'd think you'd have to jettison the `NSTextField` instance and switch to a different widget like `NSTextView`. Don't leap to conclusions and dump the existing text field, however, because the Cocoa `NSTextField` widget supports the rich-text editing that our vision for the equation editor requires. We could certainly switch to a `NSTextView` control, but that control is designed for multiple-line entry. `NSTextField` is for single-line entry and supports all of the colors, fonts, and sizes we need.

The trick to changing colors, fonts, and sizes in the text field revolves around what Cocoa calls an *attributed string*, embodied in the `NSAttributedString` class, which is included in every `NSControl` instance. As its name indicates, an attributed string is a string with attributes—attributes such as color or size across specified ranges of text. To unveil the attributed string in a text field, either you must call `setAllowsEditingTextAttributes:YES` on the `NSTextField` instance or you must turn on **Allows Rich Text** in Interface Builder, as shown in Figure 4–15. Otherwise, any attributes you specify don't display, contrary to what the documentation for `NSTextField`'s `setAllowsEditingTextAttributes:` suggests.

Turn on attributes for Graphique's equation entry field by selecting `EquationEntryViewController.xib` to open it in Interface Builder, selecting the equation text field, and checking **Allows Rich Text** in the Attributes inspector. By checking that box, we've transformed the plain and boring text field into a field capable of displaying all the cool attributes we have planned for it.

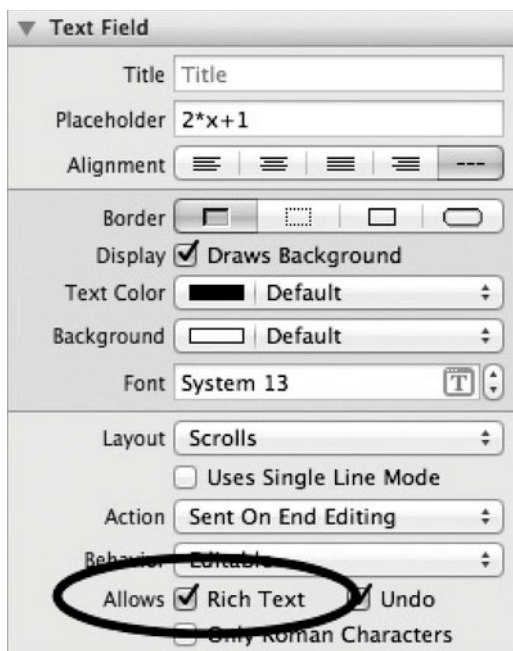


Figure 4–15. Setting the *NSTextField* to show rich text

Creating the Tokens

Most of the planned functionality for our improved equation entry editor relies on knowing what the individual pieces of the entered equation are—not just the individual characters but the groups of characters that go together and what they mean. To do this, we must parse or tokenize the equation into its individual tokens and then make decisions about things such as syntax coloring or implicit multiplication using those tokens. The kinds of tokens we support are as follows:

- Numbers
- Variables (only *x*)
- Operators (for example, + or -)
- Open parentheses
- Close parentheses
- Exponents
- Symbols
- Trigonometric functions (sine and cosine)
- Spaces
- Invalid input

Our strategy is to define an enum with these token types and then a token class that holds the token's type and its value. If, for example, the user enters the number 123, we create an instance of this token class that holds the token's type, which is a number, and the token's value, which is 123. In addition, the token class will hold a boolean field that stores whether this token is valid, since a token can be an valid type yet still be invalid, as in the case of an open parenthesis without a matching close parenthesis.

Create a new Cocoa Objective-C class called `EquationToken` as a subclass of `NSObject`. `EquationToken.h`, shown in Listing 4–8, creates the enum for the token types and declares properties for the token's type, value, and validity, as well as an initializer method that takes a type and value. You can see that the names for the token types unmistakably describe what they represent, which is a good practice that not only makes your code more understandable and maintainable but also lets you shirk documentation in most cases.

Listing 4–8. *EquationToken.h*

```
#import <Foundation/Foundation.h>

typedef enum
{
    EquationTokenTypeInvalid = 0,
    EquationTokenTypeNumber,
    EquationTokenTypeVariable,
    EquationTokenTypeOperator,
    EquationTokenTypeOpenParen,
    EquationTokenTypeCloseParen,
    EquationTokenTypeExponent,
    EquationTokenTypeSymbol,
    EquationTokenTypeTrigFunction,
    EquationTokenTypeWhitespace,
} EquationTokenType;

@interface EquationToken : NSObject

@property (nonatomic) EquationTokenType type;
@property (nonatomic, retain) NSString *value;
@property (nonatomic) BOOL valid;

- (id)initWithType:(EquationTokenType)type andValue:(NSString *)value;

@end
```

`EquationToken.m`, shown in Listing 4–9, handles the getting and setting of these properties. It also initializes each token as valid, except for tokens of type `EquationTokenTypeInvalid`.

Listing 4–9. *EquationToken.m*

```
#import "EquationToken.h"

@implementation EquationToken

@synthesize type;
```

```

@synthesize value;
@synthesize valid;

- (id)initWithType:(EquationTokenType)type_ andValue:(NSString *)value_
{
    self = [super init];
    if (self)
    {
        self.type = type_;
        self.value = value_;
        self.valid = (type_ != EquationTokenTypeInvalid);
    }
    return self;
}

@end

```

Building a class to store tokens is the easier half of the task to tokenize equations. Next, we have to actually parse or tokenize the equation, which we do next.

Parsing the Equation

We could build an `EquationTokenizer` class to tokenize the equation, but designing with fat models—a term that refers to model classes that also have the smarts to manipulate themselves, rather than rely on controller classes to manipulate them—seems more appropriate. In that vein, we'll build the tokenizing of an equation into the `Equation` class itself.

Before we start parsing the equation, however, we set up three arrays that hold special strings that we look for when parsing equations. These arrays hold the operators, the trigonometric functions, and the symbols we support, and we use them in our parsing routine to detect operators, trigonometric functions, and symbols. Create them as static arrays inside `Equation.m`, between the class extension and the start of the implementation, as shown in Listing 4–10.

Listing 4–10. *Declaring the Arrays in `Equation.m`*

```

...
@interface Equation ()
- (BOOL)produceError:(NSError**)error withCode:(NSInteger)code
andMessage:(NSString*)message;
@end

static NSArray *OPERATORS;
static NSArray *TRIG_FUNCTIONS;
static NSArray *SYMBOLS;

@implementation Equation
...

```

Initialize their contents inside the `initialize:` method, which is a class method that is called when a class loads. The `OPERATORS` array contains the operators we support: `+`, `-`, `*`, `/`, and `^`. The `TRIG_FUNCTIONS` array contains the trigonometric functions we support:

sine (sin) and cosine (cos). Finally, the SYMBOLS array contains the symbols we support: pi (both as the two character “pi” and the single character pi symbol, typed using Option+p) and e. Listing 4–11 shows the implementation of initialize:.

Listing 4–11. *The initialize: Method in Equation.m*

```
+ (void)initialize
{
    OPERATORS = [NSArray arrayWithObjects:@"+", @"-", @"*", @"/", @"^", nil];
    TRIG_FUNCTIONS = [NSArray arrayWithObjects:@"sin", @"cos", nil];
    SYMBOLS = [NSArray arrayWithObjects:@"pi", @"e", @"\u03c0", nil];
}
```

You’ll see these arrays used later as you develop the method to parse and tokenize equations.

Now, declare a property in the Equation class called tokens that will hold all the parsed tokens. When an Equation object is initialized using the initWithString: method, we’ll tokenize the equation’s string and store it in the publicly accessible tokens array. Listing 4–12 shows the property in Equation.h.

Listing 4–12. *Equation.h with a tokens Property*

```
#import <Foundation/Foundation.h>

@interface Equation : NSObject
{
    @private
    NSString *text;
}
@property (nonatomic, strong) NSString *text;
@property (nonatomic, strong) NSMutableArray *tokens;

- (id)initWithString:(NSString*)string;
- (float)evaluateForX:(float)x;
- (BOOL)validate:(NSError **)error;

@end
```

You also must add a synthesize directive in Equation.m, like this:

```
@synthesize tokens;
```

Next, you add a method to do the tokenizing. This method will parse through the current equation, break it into its tokens, create the corresponding EquationToken instances, and put them in the tokens array. The declaration for the tokenizing method, which you put in the class extension in Equation.m, is shown in Listing 4–13.

Listing 4–13. *Declaring the tokenize: Method*

```
@interface Equation ()
- (BOOL)produceError:(NSError**)error withCode:(NSInteger)code
andMessage:(NSString*)message;
- (void)tokenize;
@end
```

Edit the `initWithString:` method to create the `tokens` array, and then call the `tokenize:` method, as shown in Listing 4–14.

Listing 4–14. *Initializing tokens*

```
- (id)initWithString:(NSString *)string
{
    self = [super init];
    if (self)
    {
        self.text = string;
        self.tokens = [NSMutableArray array];
        [self tokenize];
    }
    return self;
}
```

Implementing the Method to Tokenize the Equation

Now, turn your attention to implementing the `tokenize:` method. We have some serious work to do. We must walk through the equations that users enter and turn them into tokens of one or more characters. Some of this gets a little messy, so we build the method for tokenizing the string in several steps through the next several sections, adding more parsing capabilities as we go.

At its core, tokenizing the equation means going through its characters, one by one, and creating tokens that represent the characters' types and values. Because some tokens can comprise multiple characters, however, we can't just walk through the string of characters one by one. Numbers, for example, can have multiple digits, and we should create a single token to represent the number 100, rather than three tokens (one for "1" and one for each "0"). Instead, we must read ahead into the string to create multicharacter tokens as appropriate. With that in mind, first import the `EquationToken.h` header into `Equation.m`:

```
#import "EquationToken.h"
```

Then, start creating the `tokenize:` method with the code in Listing 4–15.

Listing 4–15. *The Initial tokenize: Method*

```
- (void)tokenize
{
    [tokens removeAllObjects];
    NSString *temp = @"";
    EquationToken *token = nil;
    for (NSUInteger i = 0, n = text.length; i < n; i++)
    {
        unichar c = [text characterAtIndex:i];
        temp = [temp stringByAppendingFormat:@"%C", c];

        // Something goes here

        [tokens addObject:token];
        temp = @"";
    }
}
```



```

    }
}

```

This code creates an array to store all the tokens it will create and creates a temporary string variable to hold the one or more characters that represent the token. It creates a pointer variable to a token that it will use to temporarily hold the current token, before adding it to the array. Then, the code loops through the input string, stored in the member variable `text`, a character at a time, appending each character to the temp string. It clears the temp string before looping. This code is clean and straightforward, but it doesn't yet accomplish anything useful. See the comment "Something goes here"? We still must add code to recognize the characters being parsed and create the appropriate token.

Converting a String to a Token

To create the token, create a new method called `newTokenFromString:` that accepts a string and returns a corresponding token, with the corresponding token type and value. Since this method shouldn't belong to the public interface (no other code should create tokens), add the declaration to the class extension, as shown in Listing 4-16.

Listing 4-16. *Declaring `newTokenFromString:`*

```

@interface Equation ()
- (BOOL)produceError:(NSError**)error withCode:(NSInteger)code
andMessage:(NSString*)message;
- (void)tokenize;
- (EquationToken *)newTokenFromString:(NSString *)string;
@end

```

The implementation of `newTokenFromString:` creates a lowercase version of the string and then compares it with various known token values to determine what type of token to create. Remember, for example, the arrays we created for the operators, trigonometric functions, and symbols we support? We compare the specified string to the contents of these arrays to determine whether to create a token of one of those types. We also check for parentheses, variables (the letter "x"), numbers, or spaces. If the passed value matches none of these, we recognize it as an invalid token. Listing 4-17 shows the implementation for `newTokenFromString:`.

Listing 4-17. *`newTokenFromString:`*

```

- (EquationToken *)newTokenFromString:(NSString *)string
{
    EquationTokenType type;
    string = [string lowercaseString];
    if ([OPERATORS containsObject:string])
    {
        type = EquationTokenTypeOperator;
    }
    else if ([TRIG_FUNCTIONS containsObject:string])
    {
        type = EquationTokenTypeTrigFunction;
    }
}

```

```

else if ([SYMBOLS containsObject:string])
{
    type = EquationTokenTypeSymbol;
}
else if ([string isEqualToString:@"("])
{
    type = EquationTokenTypeOpenParen;
}
else if ([string isEqualToString:@")"])
{
    type = EquationTokenTypeCloseParen;
}
else if ([string isEqualToString:@"x"])
{
    type = EquationTokenTypeVariable;
}
// Digits are all grouped together in the tokenize: method, so just check the first
character
else if (isdigit([string characterAtIndex:0]) || [string characterAtIndex:0] == '.')
{
    type = EquationTokenTypeNumber;
}
// Spaces are all grouped together in the tokenize: method, so just check the first
character
else if ([string characterAtIndex:0] == ' ')
{
    type = EquationTokenTypeWhitespace;
}
else
{
    type = EquationTokenTypeInvalid;
}
return [[EquationToken alloc] initWithType:type andValue:string];
}

```

Having a method to create the appropriate token from a string allows us to update the “Something goes here” comment in the `tokenize:` method to actually create the token, so remove the comment and add a call to `newTokenFromString:`, as in Listing 4–18.

Listing 4–18. *Calling `newTokenFromString:` from the `tokenize:` Method*

```

- (void)tokenize
{
    [tokens removeAllObjects];
    NSString *temp = @"";
    EquationToken *token = nil;
    for (NSUInteger i = 0, n = text.length; i < n; i++)
    {
        unichar c = [text characterAtIndex:i];
        temp = [temp stringByAppendingFormat:@"%C", c];

        token = [self newTokenFromString:temp];

        [tokens addObject:token];
        temp = @"";
    }
}

```

```
}
}
```

This is a good start; this implementation of `tokenize`: recognizes single-character tokens accurately: parentheses, variables, `e`, the single-character version of `pi`, operators, and numbers that happen to be a single digit. We're not ready, however, to release this version, or even test it, until we add support for multiple-character tokens. Although recognizing each kind of multiple-character token shares similar approaches, each requires some special handling that requires different code. We tackle each of these scenarios in turn, sacrificing some code optimization opportunities for the purpose of maintaining clarity.

Recognizing Numbers

As we walk through the characters of the input string, when we come across a digit character we want to continue to read characters and tack them onto our `temp` string until we come to the first nondigit character (or the end of the string). This way, we contain a valid, multidigit number in a single token. We also want to support decimals, so we treat a period like a digit. Our code for recognizing multidigit numbers is shown in Listing 4–19.

Listing 4–19. *Recognizing Multidigit Numbers*

```
- (void)tokenize
{
    [tokens removeAllObjects];
    NSString *temp = @"";
    EquationToken *token = nil;
    for (NSUInteger i = 0, n = text.length; i < n; i++)
    {
        unichar c = [text characterAtIndex:i];
        temp = [temp stringByAppendingFormat:@"%C", c];

        // Keep all digits of a number as one token
        if (isdigit(c) || c == '.')
        {
            // Keep reading characters until we hit the end of the string
            while (i < (n - 1))
            {
                // Increment our loop variable
                ++i;
                // Get the next character
                c = [text characterAtIndex:i];
                // Test to see whether to continue
                if (isdigit(c) || c == '.')
                {
                    // Append the character to the temp string
                    temp = [temp stringByAppendingFormat:@"%C", c];
                }
            }
        }
        else
        {
            // Character didn't match, so back the loop counter up and exit
            --i;
        }
    }
}
```

```

        break;
    }
}

token = [self newTokenFromString:temp];
[tokens addObject:token];
temp = @"";
}
}

```

Notice that if we get to a nondigit or nonperiod character, we back up the loop counter by one so that the character we read isn't lost but rather is read the next time through the loop as we start reading a new token.

You might also notice that this code allows a number to have more than one decimal point, which shouldn't be valid. Rather than complicate this code with decimal point counting, however, we'll catch multiple decimal points later.

Grouping Spaces

Although some people can comfortably squash their equations together, with numbers, operators, parentheses, and any other characters mashed together without any spaces, others like their equations to breathe a little, inserting spaces between numbers and operators to increase readability. We shouldn't care about spaces and should treat these three equations identically:

```

x^2+2*x+7
x^2 + 2*x + 7
x ^ 2  +  2 * x  + 7

```

We should also lump multiple consecutive spaces into a single token with the type `EquationTokenTypeWhitespace`. To combine consecutive spaces, we can employ an algorithm that's nearly identical to our digit detection code. Listing 4–20 shows the addition of code to detect multiple spaces.

Listing 4–20. Code to Recognize Multiple Spaces

```

- (void)tokenize
{
    [tokens removeAllObjects];
    NSString *temp = @"";
    EquationToken *token = nil;
    for (NSUInteger i = 0, n = text.length; i < n; i++)
    {
        unichar c = [text characterAtIndex:i];
        temp = [temp stringByAppendingFormat:@"%C", c];

        // Keep all digits of a number as one token
        if (isdigit(c) || c == '.')
        {
            // Keep reading characters until we hit the end of the string
            while (i < (n - 1))
            {

```

```

        // Increment our loop variable
        ++i;
        // Get the next character
        c = [text characterAtIndex:i];
        // Test to see whether to continue
        if (isdigit(c) || c == '.')
        {
            // Append the character to the temp string
            temp = [temp stringByAppendingFormat:@"%C", c];
        }
        else
        {
            // Character didn't match, so back the loop counter up and exit
            --i;
            break;
        }
    }
}

// Keep all spaces together
else if (c == ' ')
{
    // Keep reading characters until we hit the end of the string
    while (i < (n - 1))
    {
        // Increment our loop variable
        ++i;
        // Get the next character
        c = [text characterAtIndex:i];
        // Test to see whether to continue
        if (c == ' ')
        {
            // Append the character to the temp string
            temp = [temp stringByAppendingFormat:@"%C", c];
        }
        else
        {
            // Character didn't match, so back the loop counter up and exit
            --i;
            break;
        }
    }
}

token = [self newTokenFromString:temp];

[tokens addObject:token];
temp = @"";
}
}

```

Read through this code and note the similarities to the digit detection code. We could try to do something clever to reduce the code duplication—Objective-C's relatively new support for blocks comes to mind—but we'll leave the code as is to keep things relatively simple.

We've now written code to recognize multicharacter digits and multiple consecutive spaces. We have two other multiple character token possibilities, however: trigonometric functions and symbols. We knock out those two token types in the next section.

Recognizing Trigonometric Functions and Symbols

Remember the arrays we created, called `TRIG_FUNCTIONS` and `SYMBOLS`, to hold the trigonometric functions and symbols we support? We're ready to use them now to detect whether the current token belongs to one of these groups. The approach for both these arrays is the same:

1. Loop through the elements in the array.
2. For each element, determine whether its length could fit in what remains of the input string.
3. If the length fits, compare the element to the input string to see whether they match.
4. If they match, move the index counter to the end of the matching string.

To compare the element to a piece of the input string, we use `NSString`'s `substringWithRange:` method, which grabs a piece from the string using the specified range. Ranges specify a starting index and a length. To create the range, we use the `NSMakeRange` function, which takes an index and a length and returns an `NSRange` struct. The index we use is the current index, `i`, and the length is the length of the element we're comparing the input string to. The code to get the substring looks like this:

```
[text substringWithRange:NSMakeRange(i, [element length])];
```

One more thing we want to do when making the comparison: we want to normalize the case of the text so that both `sin` and `SIN`, for example, match `sin` (the sine trigonometric function). We use `NSString`'s `lowercaseString:` method to convert the input substring to lowercase, which is the case we used for the elements in both `TRIG_FUNCTIONS` and `SYMBOLS`.

Listing 4–21 shows the growing `tokenize:` method with the code added to detect trigonometric functions and symbols.

Listing 4–21. Detecting Trigonometric Functions and Symbols

```
- (void)tokenize
{
    [tokens removeAllObjects];
    NSString *temp = @"";
    EquationToken *token = nil;
    for (NSUInteger i = 0, n = text.length; i < n; i++)
    {
        unichar c = [text characterAtIndex:i];
        temp = [temp stringByAppendingFormat:@"%C", c];

        // Keep all digits of a number as one token
        if (isdigit(c) || c == '.')
    }
```

```

{
    // Keep reading characters until we hit the end of the string
    while (i < (n - 1))
    {
        // Increment our loop variable
        ++i;
        // Get the next character
        c = [text characterAtIndex:i];
        // Test to see whether to continue
        if (isdigit(c) || c == '.')
        {
            // Append the character to the temp string
            temp = [temp stringByAppendingFormat:@"%C", c];
        }
        else
        {
            // Character didn't match, so back the loop counter up and exit
            --i;
            break;
        }
    }
}

// Keep all spaces together
else if (c == ' ')
{
    // Keep reading characters until we hit the end of the string
    while (i < (n - 1))
    {
        // Increment our loop variable
        ++i;
        // Get the next character
        c = [text characterAtIndex:i];
        // Test to see whether to continue
        if (c == ' ')
        {
            // Append the character to the temp string
            temp = [temp stringByAppendingFormat:@"%C", c];
        }
        else
        {
            // Character didn't match, so back the loop counter up and exit
            --i;
            break;
        }
    }
}

// Check for trig functions
for (NSString *trig in TRIG_FUNCTIONS)
{
    if (trig.length <= (n - i) && [trig isEqualToString:[text
substringWithRange:NSMakeRange(i, trig.length)] lowercaseString])
    {
        temp = trig;
        i += (trig.length) - 1;
        break;
    }
}

```

```

    }
}

// Check for symbols
for (NSString *symbol in SYMBOLS)
{
    if (symbol.length <= (n - i) && [symbol isEqualToString:[text
substringWithRange:NSMakeRange(i, symbol.length)] lowercaseString]])
    {
        temp = symbol;
        i += (symbol.length - 1);
        break;
    }
}

token = [self newTokenFromString:temp];

[tokens addObject:token];
temp = @"";
}
}

```

At this point, we've added all the code before the line that calls `newTokenFromString:`. We're ready to let that line of code run and create an `EquationToken` object from the input. We still have some cleanup to do after that call, though, which is outlined in the next few sections.

Recognizing Exponents

So far, our tokenizing recognizes all digits as number tokens. It should recognize some of these digits, however, as exponents. We recognize the following tokens as exponents instead of numbers:

- Numbers that immediately follow the `^` symbol (for example, `^2`)
- Numbers that immediately follow a variable (for example, `x2`)
- Numbers that immediately follow a close parenthesis (for example, `(x+3)2`)

After we've created the current token, then we check to see whether we have a number token and then whether we should change it from a number type to an exponent type. To make the determination, we check whether we have a previous token and, if so, whether it matches any of the three rules listed earlier for exponents. The code to do this follows the line of code that creates the new token and is shown in Listing 4–22.

Listing 4–22. Finding Exponents

```

- (void)tokenize
{
    [tokens removeAllObjects];
    NSString *temp = @"";
    EquationToken *token = nil;
    for (NSUInteger i = 0, n = text.length; i < n; i++)

```



```

{
    unichar c = [text characterAtIndex:i];
    temp = [temp stringByAppendingFormat:@"%C", c];

    // Keep all digits of a number as one token
    if (isdigit(c) || c == '.')
    {
        // Keep reading characters until we hit the end of the string
        while (i < (n - 1))
        {
            // Increment our loop variable
            ++i;
            // Get the next character
            c = [text characterAtIndex:i];
            // Test to see whether to continue
            if (isdigit(c) || c == '.')
            {
                // Append the character to the temp string
                temp = [temp stringByAppendingFormat:@"%C", c];
            }
            else
            {
                // Character didn't match, so back the loop counter up and exit
                --i;
                break;
            }
        }
    }

    // Keep all spaces together
    else if (c == ' ')
    {
        // Keep reading characters until we hit the end of the string
        while (i < (n - 1))
        {
            // Increment our loop variable
            ++i;
            // Get the next character
            c = [text characterAtIndex:i];
            // Test to see whether to continue
            if (c == ' ')
            {
                // Append the character to the temp string
                temp = [temp stringByAppendingFormat:@"%C", c];
            }
            else
            {
                // Character didn't match, so back the loop counter up and exit
                --i;
                break;
            }
        }
    }

    // Check for trig functions
    for (NSString *trig in TRIG_FUNCTIONS)
    {

```

```

        if (trig.length <= (n - i) && [trig isEqualToString:[text
substringWithRange:NSMakeRange(i, trig.length)] lowercaseString]])
        {
            temp = trig;
            i += (trig.length - 1);
            break;
        }
    }

    // Check for symbols
    for (NSString *symbol in SYMBOLS)
    {
        if (symbol.length <= (n - i) && [symbol isEqualToString:[text
substringWithRange:NSMakeRange(i, symbol.length)] lowercaseString]])
        {
            temp = symbol;
            i += (symbol.length - 1);
            break;
        }
    }

    token = [self newTokenFromString:temp];

    // Determine if this should be an exponent
    // Check that we have a previous token to follow and that this is a number
    if (token.type == EquationTokenTypeNumber && !(tokens.count == 0))
    {
        // Get the previous token
        EquationToken *previousToken = [tokens lastObject];

        // If the previous token is a variable, close parenthesis, or the ^ operator, this
is an exponent
        if (previousToken.type == EquationTokenTypeVariable ||
            previousToken.type == EquationTokenTypeCloseParen ||
            [previousToken.value isEqualToString:@"^"])
        {
            token.type = EquationTokenTypeExponent;
        }
    }

    [tokens addObject:token];
    temp = @"";
}
}
}

```

You can see that this code does as we describe, changing the current token from a number type to an exponent type if any of the three exponent rules match. To verify that this code works, create a unit test for it by creating a new Cocoa Objective-C test case class, as shown in Figure 4–16, and click Next. Call the class `ExponentTests` and set the test type to Logic, as shown in Figure 4–17. Click Next, and save it in the `GraphiqueTests` folder, the `GraphiqueTests` group, and the `GraphiqueTests` target.

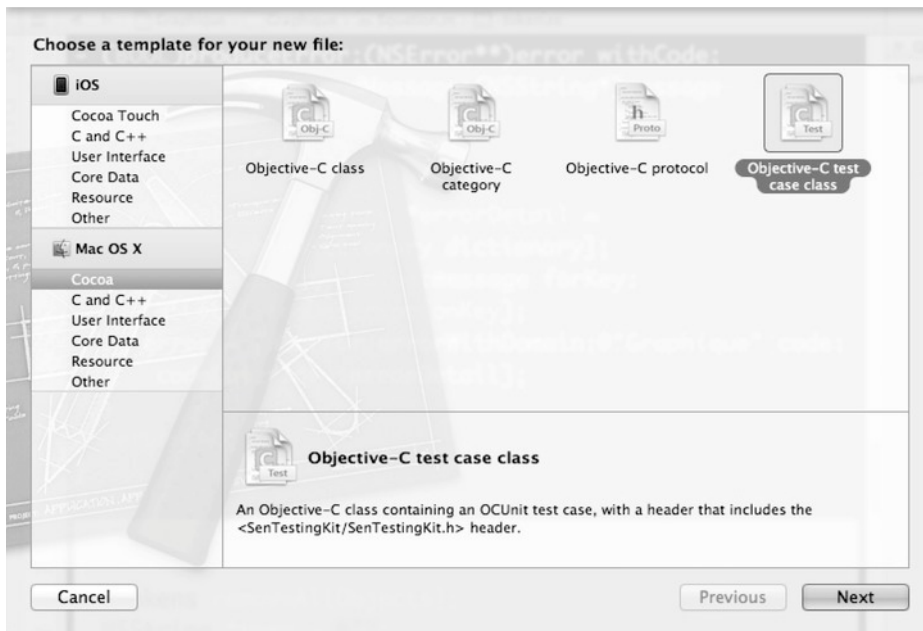


Figure 4-16. *Creating a test case*

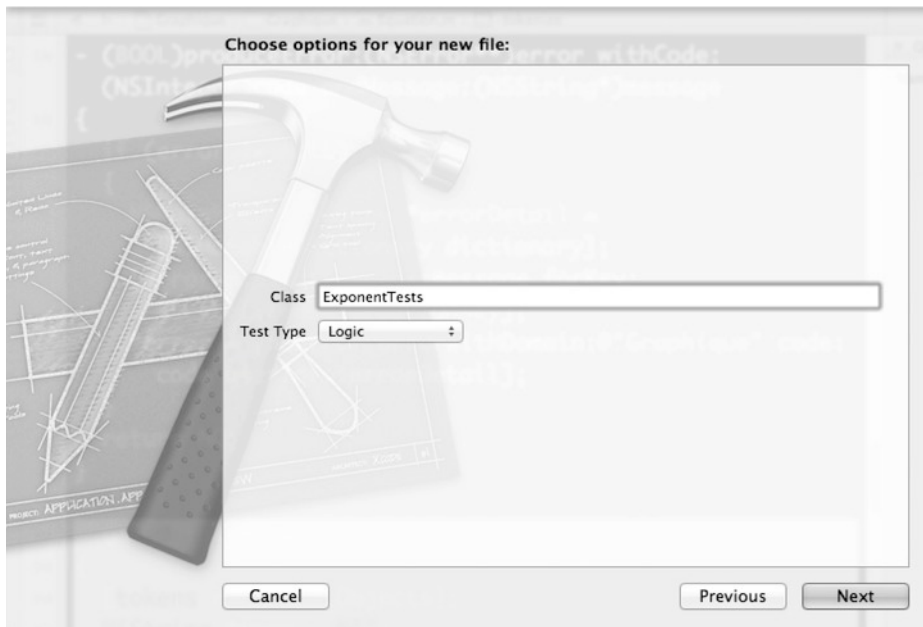


Figure 4-17. *Naming the test class and setting its type*

The header file, shown in Listing 4–23, declares a single method called `testExponents:`.

Listing 4–23. *ExponentTests.h*

```
#import <SenTestingKit/SenTestingKit.h>

@interface ExponentTests : SenTestCase

- (void)testExponents;

@end
```

In the implementation of `testExponents:`, create an equation with tokens of various types, including both numbers and exponents, and verify that the numbers stay numbers and the exponents become exponents. Listing 4–24 shows the implementation.

Listing 4–24. *ExponentTests.m*

```
#import "ExponentTests.h"
#import "Equation.h"
#import "EquationToken.h"

@implementation ExponentTests

-(void)testExponents
{
    Equation *equation = [[Equation alloc] initWithString:@"32x2+(x+7)45+3^3"];
    NSArray *tokens = equation.tokens;

    STAssertTrue(tokens.count == 14, NULL);

    EquationToken *token = nil;

    token = [tokens objectAtIndex:0];
    STAssertEquals(EquationTokenTypeNumber, token.type, NULL);
    STAssertEqualObjects(@"32", token.value, NULL);

    token = [tokens objectAtIndex:2];
    STAssertEquals(EquationTokenTypeExponent, token.type, NULL);
    STAssertEqualObjects(@"2", token.value, NULL);

    token = [tokens objectAtIndex:7];
    STAssertEquals(EquationTokenTypeNumber, token.type, NULL);
    STAssertEqualObjects(@"7", token.value, NULL);

    token = [tokens objectAtIndex:9];
    STAssertEquals(EquationTokenTypeExponent, token.type, NULL);
    STAssertEqualObjects(@"45", token.value, NULL);

    token = [tokens objectAtIndex:11];
    STAssertEquals(EquationTokenTypeNumber, token.type, NULL);
    STAssertEqualObjects(@"3", token.value, NULL);

    token = [tokens objectAtIndex:13];
    STAssertEquals(EquationTokenTypeExponent, token.type, NULL);
```

```
STAssertEqualObjects(@"3", token.value, NULL);  
}
```

```
@end
```

Run your tests using `⌘+U` and verify that they all pass. Assuming they do, you have successfully implemented exponent recognition.

Creating a Stack for Parenthesis Matching

Effectively matching parentheses turns out to be a problem with a straightforward solution, as described in such sites as www.ccs.neu.edu/home/sbratus/com1101/lab4.html. To summarize, an approach that works cleanly and simply is as follows:

1. Create a stack.
2. Read through your input, character by character.
3. If the current character is an open parenthesis, push it onto your stack.
4. If the current character is a close parenthesis and you have an open parenthesis at the top of your stack, pop the open parenthesis and match it with this close parenthesis.
5. Any close parentheses without corresponding open parentheses on the stack are unmatched.
6. Any remaining open parentheses on the stack, once you've read all the input, are unmatched.

To implement our parenthesis matching, then, we start by creating a stack. A stack stores and retrieves a collection of data in a last in, first out (LIFO) approach: a “retrieve” operation returns the most recently stored object. Storing an object in a stack is typically called *pushing* the object onto the stack, and retrieving the object from the stack is usually called *popping* it off the stack. Popping an object off the stack both retrieves it from the stack and deletes it from the stack. A typical stack class, then, supports a push and a pop operation. Some stack implementations also support a peek operation that returns the most recently stored object but doesn't delete it from the stack, and some support an operation for determining whether the stack is currently empty (contains no data). Which operations does the Cocoa stack class support? None at all—Cocoa has no stack class!

Luckily, however, writing a stack class is fairly simple, especially because you can leverage Cocoa's `NSMutableArray` class to do all the storage and retrieval. In this section, we create a stack class called `Stack` and incorporate it into the equation editor to do the parenthesis balancing.

Creating the Stack Class

Start creating your stack by creating a new Cocoa Objective-C class called `Stack` that derives from `NSObject`. This class will offer three operations:

- `push`, to push an object onto the stack
- `pop`, to pop an object off the stack
- `hasObjects`, to determine whether the stack has any objects

In addition to declaring methods for these three operations in `Stack.h`, you also must declare the `NSMutableArray` private member for storing and retrieving data. Listing 4–25 shows the code for `Stack.h`.

Listing 4–25. *Stack.h*

```
#import <Foundation/Foundation.h>

@interface Stack : NSObject {
@private
    NSMutableArray *stack;
}

- (void)push:(id)anObject;
- (id)pop;
- (BOOL)hasObjects;

@end
```

The implementation file, `Stack.m`, creates the stack `NSMutableArray` member in its `init:` method. The `Stack` class's `push:` method adds the passed object to the end of `stack`. The `pop:` method retrieves the last object from `stack`, deletes the object from `stack`, and then returns the object. Finally, the `hasObjects:` method returns whether the stack has any objects using `NSMutableArray`'s `count:` method. Listing 4–26 shows `Stack.m`.

Listing 4–26. *Stack.m*

```
#import "Stack.h"

@implementation Stack

- (id)init
{
    self = [super init];
    if (self)
    {
        stack = [NSMutableArray array];
    }
    return self;
}

- (void)push:(id)anObject
{
    [stack addObject:anObject];
}
```

```

}

- (id)pop
{
    id anObject = [stack lastObject];
    [stack removeObject:anObject];
    return anObject;
}

- (BOOL)hasObjects
{
    return [stack count] > 0;
}

@end

```

See how simple that was? The NSMutableArray class takes care of the tricky parts of storage and retrieval.

Before we incorporate Stack into the equation editor, however, we should test it to ensure it works as we expect. Read on to the next section to see how to write automated unit tests for Stack.

Testing the Stack Class

To test the Stack class, create a test case that pushes 1,000 objects onto a stack, makes sure the stack isn't empty, and then pops the objects off the stack, one at a time, and verifies that the 1,000 pushed objects are popped in reverse order. Finally, verify that after all 1,000 objects are popped off the stack, the stack is empty. Create a new Cocoa Objective-C test case class called StackTests, making it a Logic-type test and making sure to add it to the GraphiqueTests folder and the GraphiqueTests target but not the Graphique target. Declare a method called testPushAndPop: in StackTests.h so that it matches Listing 4–27.

Listing 4–27. StackTests.h

```

#import <SenTestingKit/SenTestingKit.h>

@interface StackTests : SenTestCase

- (void)testPushAndPop;

@end

```

StackTests.m implements the testPushAndPop: method as outlined earlier, pushing NSString objects that reflect their index number (0–999) in the stack so that we can easily verify that they're popped off in the correct order (999 through 0). See Listing 4–28.

Listing 4–28. StackTests.m

```

#import "StackTests.h"
#import "Stack.h"

```

```

@implementation StackTests

- (void)testPushAndPop
{
    Stack *stack = [[Stack alloc] init];
    for (int i = 0; i < 1000; i++)
    {
        [stack push:[NSString stringWithFormat:@"String #%d", i]];
    }
    STAssertTrue([stack hasObjects], @"Stack should not be empty after pushing 1,000
objects");

    for (int i = 999; i >= 0; i--)
    {
        NSString *string = (NSString *)[stack pop];
        NSString *comp = [NSString stringWithFormat:@"String #%d", i];
        STAssertEqualObjects(string, comp, NULL);
    }
    STAssertFalse([stack hasObjects], @"Stack should be empty after popping 1,000
objects");
}

@end

```

Run your tests using `⌘+U` and verify that they all pass. Now you can feel confident that Stack is ready to handle its parenthesis-balancing chores.

Balancing Parentheses Using Stack

You're ready to implement the parenthesis-matching algorithm in the `tokenize:` method. Start by importing the `Stack.h` header at the top of `Equation.m`:

```
#import "Stack.h"
```

In the `tokenize:` method, create a Stack instance. Then, after the exponent-detection code, include code that uses the Stack instance to perform parenthesis matching. This code should set each new open parenthesis as invalid, because it's not yet matched, and set it to valid only when it's matched to a close parenthesis. Listing 4–29 contains the updated `tokenize:` method.

Listing 4–29. *Parenthesis Matching*

```

- (void)tokenize
{
    [tokens removeAllObjects];
    Stack *stack = [[Stack alloc] init];

    NSString *temp = @"";
    EquationToken *token = nil;
    for (NSUInteger i = 0, n = text.length; i < n; i++)
    {
        unichar c = [text characterAtIndex:i];
        temp = [temp stringByAppendingFormat:@"%C", c];

        // Keep all digits of a number as one token
    }
}

```



```

if (isdigit(c) || c == '.')
{
    // Keep reading characters until we hit the end of the string
    while (i < (n - 1))
    {
        // Increment our loop variable
        ++i;
        // Get the next character
        c = [text characterAtIndex:i];
        // Test to see whether to continue
        if (isdigit(c) || c == '.')
        {
            // Append the character to the temp string
            temp = [temp stringByAppendingFormat:@"%C", c];
        }
        else
        {
            // Character didn't match, so back the loop counter up and exit
            --i;
            break;
        }
    }
}

// Keep all spaces together
else if (c == ' ')
{
    // Keep reading characters until we hit the end of the string
    while (i < (n - 1))
    {
        // Increment our loop variable
        ++i;
        // Get the next character
        c = [text characterAtIndex:i];
        // Test to see whether to continue
        if (c == ' ')
        {
            // Append the character to the temp string
            temp = [temp stringByAppendingFormat:@"%C", c];
        }
        else
        {
            // Character didn't match, so back the loop counter up and exit
            --i;
            break;
        }
    }
}

// Check for trig functions
for (NSString *trig in TRIG_FUNCTIONS)
{
    if (trig.length <= (n - i) && [trig isEqualToString:[text
substringWithRange:NSMakeRange(i, trig.length)] lowercaseString]])
    {
        temp = trig;
        i += (trig.length - 1);
    }
}

```

```

        break;
    }
}

// Check for symbols
for (NSString *symbol in SYMBOLS)
{
    if (symbol.length <= (n - i) && [symbol isEqualToString:[text
substringWithRange:NSMakeRange(i, symbol.length)] lowercaseString]])
    {
        temp = symbol;
        i += (symbol.length - 1);
        break;
    }
}

token = [self newTokenFromString:temp];

// Determine if this should be an exponent
// Check that we have a previous token to follow and that this is a number
if (token.type == EquationTokenTypeNumber && !(tokens.count == 0))
{
    // Get the previous token
    EquationToken *previousToken = [tokens lastObject];

    // If the previous token is a variable, close parenthesis, or the ^ operator, this
is an exponent
    if (previousToken.type == EquationTokenTypeVariable ||
        previousToken.type == EquationTokenTypeCloseParen ||
        [previousToken.value isEqualToString:@"^"])
    {
        token.type = EquationTokenTypeExponent;
    }
}

// Do parenthesis matching
if (token.type == EquationTokenTypeOpenParen)
{
    // Set the new open parenthesis to invalid, as it's not yet matched,
    // and push it onto the stack
    token.valid = NO;
    [stack push:token];
}
else if (token.type == EquationTokenTypeCloseParen)
{
    // See if we have a matching open parenthesis
    if (![stack hasObjects])
    {
        // No open parenthesis to match, so this close parenthesis is invalid
        token.valid = NO;
    }
    else
    {
        // We have a matching open parenthesis, so set it (and this close parenthesis)
        // to valid and pop the open parenthesis off the stack
        EquationToken *match = [stack pop];
    }
}

```

```

        match.valid = YES;
    }
}

[tokens addObject:token];
temp = @"";
}
}

```

Detecting Multiple Decimal Points

We said before that you still must catch numbers with multiple decimal points and mark them as invalid. To do this, add code that splits any number tokens into components separated by decimal points. If you have more than two such components (the number before the decimal point and the number after), the token is invalid. Listing 4–30 shows the `tokenize:` method with this code added.

Listing 4–30. *Detecting Multiple Decimal Points*

```

- (void)tokenize
{
    [tokens removeAllObjects];
    Stack *stack = [[Stack alloc] init];

    NSString *temp = @"";
    EquationToken *token = nil;
    for (NSUInteger i = 0, n = text.length; i < n; i++)
    {
        unichar c = [text characterAtIndex:i];
        temp = [temp stringByAppendingFormat:@"%C", c];

        // Keep all digits of a number as one token
        if (isdigit(c) || c == '.')
        {
            // Keep reading characters until we hit the end of the string
            while (i < (n - 1))
            {
                // Increment our loop variable
                ++i;
                // Get the next character
                c = [text characterAtIndex:i];
                // Test to see whether to continue
                if (isdigit(c) || c == '.')
                {
                    // Append the character to the temp string
                    temp = [temp stringByAppendingFormat:@"%C", c];
                }
                else
                {
                    // Character didn't match, so back the loop counter up and exit
                    --i;
                    break;
                }
            }
        }
    }
}

```

```

// Keep all spaces together
else if (c == ' ')
{
    // Keep reading characters until we hit the end of the string
    while (i < (n - 1))
    {
        // Increment our loop variable
        ++i;
        // Get the next character
        c = [text characterAtIndex:i];
        // Test to see whether to continue
        if (c == ' ')
        {
            // Append the character to the temp string
            temp = [temp stringByAppendingFormat:@"%C", c];
        }
        else
        {
            // Character didn't match, so back the loop counter up and exit
            --i;
            break;
        }
    }
}

// Check for trig functions
for (NSString *trig in TRIG_FUNCTIONS)
{
    if (trig.length <= (n - i) && [trig isEqualToString:[text
substringWithRange:NSMakeRange(i, trig.length)] lowercaseString]])
    {
        temp = trig;
        i += (trig.length - 1);
        break;
    }
}

// Check for symbols
for (NSString *symbol in SYMBOLS)
{
    if (symbol.length <= (n - i) && [symbol isEqualToString:[text
substringWithRange:NSMakeRange(i, symbol.length)] lowercaseString]])
    {
        temp = symbol;
        i += (symbol.length - 1);
        break;
    }
}

token = [self newTokenFromString:temp];

// Determine if this should be an exponent
// Check that we have a previous token to follow and that this is a number
if (token.type == EquationTokenTypeNumber && !(tokens.count == 0))
{
    // Get the previous token

```

```

EquationToken *previousToken = [tokens lastObject];

// If the previous token is a variable, close parenthesis, or the ^ operator, this
is an exponent
if (previousToken.type == EquationTokenTypeVariable ||
    previousToken.type == EquationTokenTypeCloseParen ||
    [previousToken.value isEqualToString:@"^"])
{
    token.type = EquationTokenTypeExponent;
}

// Do parenthesis matching
if (token.type == EquationTokenTypeOpenParen)
{
    // Set the new open parenthesis to invalid, as it's not yet matched,
    // and push it onto the stack
    token.valid = NO;
    [stack push:token];
}
else if (token.type == EquationTokenTypeCloseParen)
{
    // See if we have a matching open parenthesis
    if (![stack hasObjects])
    {
        // No open parenthesis to match, so this close parenthesis is invalid
        token.valid = NO;
    }
    else
    {
        // We have a matching open parenthesis, so set it (and this close parenthesis)
        // to valid and pop the open parenthesis off the stack
        EquationToken *match = [stack pop];
        match.valid = YES;
    }
}

// Numbers with more than one decimal point are invalid
if (token.type == EquationTokenTypeNumber && [[token.value
componentsSeparatedByString:@"."] count] > 2)
{
    token.valid = NO;
}

[tokens addObject:token];
temp = @"";
}
}

```

The tokenize: method is now complete.

Before congratulating yourself too much, however, you should test it. You've already tested the code for the exponent recognition and the stack. In the next section, you write tests to exercise the rest of the tokenize: method.

Testing the Tokenizer

Create a new Cocoa Objective-C test case class called `EquationTokenizeTests`, with the Logic test type, and put it in the `GraphiqueTests` folder, group, and target. Declare all the test methods in `EquationTokenizeTests.h` that you'll create in this chapter, as well as a helper method that you'll use to test both the type and the value of a specified token. Listing 4–31 shows the code for `EquationTokenizeTests.h`.

Listing 4–31. *EquationTokenizeTests.h*

```
#import <SenTestingKit/SenTestingKit.h>
#import "EquationToken.h"

@interface EquationTokenizeTests : SenTestCase

- (void)testSimple;
- (void)testExponent;
- (void)testExponentWithCaret;
- (void)testExponentWithParens;
- (void)testWhitespace;
- (void)testTrigFunctionsAndSymbols;
- (void)testParenthesisMatching;
- (void)testInvalid;
- (void)helperTestToken:(EquationToken *)token type:(EquationTokenType)type
  value:(NSString *)value;

@end
```

In `EquationTokenizeTests.m`, implement the `helperTestToken:type:value` method to test the specified token's type and value against the specified type and value. Listing 4–32 shows the `EquationTokenizeTests.m` file.

Listing 4–32. *EquationTokenizeTests.m*

```
#import "EquationTokenizeTests.h"
#import "Equation.h"

@implementation EquationTokenizeTests

- (void)helperTestToken:(EquationToken *)token type:(EquationTokenType)type
  value:(NSString *)value
{
    STAssertEquals(token.type, type, NULL);
    STAssertEqualObjects(token.value, value, NULL);
}

@end
```

Now we're ready to write tests. The next few sections add tests, one at a time. Each of the tests creates an equation, parses it into its tokens, and verifies the number of tokens. It also verifies that each token has the expected type and value.

Testing Simple Equations

Start by testing a simple equation: $22 \times x - 1$. This equation should parse into five tokens:

- The number 22
- The multiplication operator
- The variable x
- The subtraction operator
- The number 1

Listing 4–33 shows the method.

Listing 4–33. *Testing a Simple Equation*

```
- (void)testSimple
{
    Equation *equation = [[Equation alloc] initWithString:@"22*x-1"];
    NSArray *tokens = equation.tokens;
    STAssertTrue(tokens.count == 5, NULL);
    [self helperTestToken:[tokens objectAtIndex:0] type:EquationTokenTypeNumber
    value:@"22"];
    [self helperTestToken:[tokens objectAtIndex:1] type:EquationTokenTypeOperator
    value:@"*"];
    [self helperTestToken:[tokens objectAtIndex:2] type:EquationTokenTypeVariable
    value:@"x"];
    [self helperTestToken:[tokens objectAtIndex:3] type:EquationTokenTypeOperator
    value:@"-"];
    [self helperTestToken:[tokens objectAtIndex:4] type:EquationTokenTypeNumber
    value:@"1"];
}
```

Testing Exponents

The next three methods test exponents. The first, `testExponent:`, tests the case of an exponent following a variable, `x`. The next, `testExponentWithCaret:`, tests the case of an exponent in the more traditional location: after a caret. Finally, `testExponentWithParens:` tests that an exponent is detected when it follows a close parenthesis. Add the code in Listing 4–34 to `EquationTokenizeTests.m`.

Listing 4–34. *Testing Exponents*

```
- (void)testExponent
{
    Equation *equation = [[Equation alloc] initWithString:@"x2"];
    NSArray *tokens = equation.tokens;
    STAssertTrue(tokens.count == 2, NULL);
    [self helperTestToken:[tokens objectAtIndex:0] type:EquationTokenTypeVariable
    value:@"x"];
    [self helperTestToken:[tokens objectAtIndex:1] type:EquationTokenTypeExponent
    value:@"2"];
}
```

```

}

- (void)testExponentWithCaret
{
    Equation *equation = [[Equation alloc] initWithString:@"x^2"];
    NSArray *tokens = equation.tokens;
    STAssertTrue(tokens.count == 3, NULL);
    [self helperTestToken:[tokens objectAtIndex:0] type:EquationTokenTypeVariable
value:@"x"];
    [self helperTestToken:[tokens objectAtIndex:1] type:EquationTokenTypeOperator
value:@"^"];
    [self helperTestToken:[tokens objectAtIndex:2] type:EquationTokenTypeExponent
value:@"2"];
}

- (void)testExponentWithParens
{
    Equation *equation = [[Equation alloc] initWithString:@"(3x+7)2"];
    NSArray *tokens = equation.tokens;
    STAssertTrue(tokens.count == 7, NULL);
    [self helperTestToken:[tokens objectAtIndex:0] type:EquationTokenTypeOpenParen
value:@"("];
    [self helperTestToken:[tokens objectAtIndex:1] type:EquationTokenTypeNumber
value:@"3"];
    [self helperTestToken:[tokens objectAtIndex:2] type:EquationTokenTypeVariable
value:@"x"];
    [self helperTestToken:[tokens objectAtIndex:3] type:EquationTokenTypeOperator
value:@"+"];
    [self helperTestToken:[tokens objectAtIndex:4] type:EquationTokenTypeNumber
value:@"7"];
    [self helperTestToken:[tokens objectAtIndex:5] type:EquationTokenTypeCloseParen
value:@")"];
    [self helperTestToken:[tokens objectAtIndex:6] type:EquationTokenTypeExponent
value:@"2"];
}

```

Testing Whitespace

We want to verify that spaces are collapsed into a single token, so we write a test that uses an equation that has multiple consecutive spaces. Listing 4–35 contains the whitespace test.

Listing 4–35. *Testing Whitespace*

```

- (void)testWhitespace
{
    Equation *equation = [[Equation alloc] initWithString:@"x  +  7"];
    NSArray *tokens = equation.tokens;
    STAssertTrue(tokens.count == 5, NULL);
    [self helperTestToken:[tokens objectAtIndex:0] type:EquationTokenTypeVariable
value:@"x"];
    [self helperTestToken:[tokens objectAtIndex:1] type:EquationTokenTypeWhitespace
value:@"  "];
}

```



```

    [self helperTestToken:[tokens objectAtIndex:2] type:EquationTokenTypeOperator
value:@"+"];
    [self helperTestToken:[tokens objectAtIndex:3] type:EquationTokenTypeWhitespace
value:@" "];
    [self helperTestToken:[tokens objectAtIndex:4] type:EquationTokenTypeNumber
value:@"7"];
}

```

Testing Trigonometric Functions and Symbols

Listing 4–36 contains the `testTrigFunctionsAndSymbols:` method, which tests for sine, cosine, pi (both as “pi” and π), and e.

Listing 4–36. *Testing Trigonometric Functions and Symbols*

```

- (void)testTrigFunctionsAndSymbols
{
    Equation *equation = [[Equation alloc]
initWithString:@"sin(0.3)+cos(3.3)+pi+e+\u03c0"];
    NSArray *tokens = equation.tokens;
    STAssertTrue(tokens.count == 15, NULL);
    [self helperTestToken:[tokens objectAtIndex:0] type:EquationTokenTypeTrigFunction
value:@"sin"];
    [self helperTestToken:[tokens objectAtIndex:1] type:EquationTokenTypeOpenParen
value:@"("];
    [self helperTestToken:[tokens objectAtIndex:2] type:EquationTokenTypeNumber
value:@"0.3"];
    [self helperTestToken:[tokens objectAtIndex:3] type:EquationTokenTypeCloseParen
value:@")"];
    [self helperTestToken:[tokens objectAtIndex:4] type:EquationTokenTypeOperator
value:@"+"];
    [self helperTestToken:[tokens objectAtIndex:5] type:EquationTokenTypeTrigFunction
value:@"cos"];
    [self helperTestToken:[tokens objectAtIndex:6] type:EquationTokenTypeOpenParen
value:@"("];
    [self helperTestToken:[tokens objectAtIndex:7] type:EquationTokenTypeNumber
value:@"3.3"];
    [self helperTestToken:[tokens objectAtIndex:8] type:EquationTokenTypeCloseParen
value:@")"];
    [self helperTestToken:[tokens objectAtIndex:9] type:EquationTokenTypeOperator
value:@"+"];
    [self helperTestToken:[tokens objectAtIndex:10] type:EquationTokenTypeSymbol
value:@"pi"];
    [self helperTestToken:[tokens objectAtIndex:11] type:EquationTokenTypeOperator
value:@"+"];
    [self helperTestToken:[tokens objectAtIndex:12] type:EquationTokenTypeSymbol
value:@"e"];
    [self helperTestToken:[tokens objectAtIndex:13] type:EquationTokenTypeOperator
value:@"+"];
    [self helperTestToken:[tokens objectAtIndex:14] type:EquationTokenTypeSymbol
value:@"\u03c0"];
}

```

Testing Parenthesis Matching

We already wrote code to test our stack implementation that the parenthesis matching uses, but you also should test the parenthesis matching directly. You already know from previous tests that you can detect parentheses as tokens, so this test doesn't use the `helperTestToken:type:value:` method. Instead, it tests the validity of the parenthesis tokens, including tests for both valid and invalid parentheses. It's in Listing 4–37.

Listing 4–37. *Testing Parenthesis Matching*

```
- (void)testParenthesisMatching
{
    {
        Equation *equation = [[Equation alloc] initWithString:@"()"];
        NSArray *tokens = equation.tokens;
        STAssertTrue(tokens.count == 2, NULL);
        STAssertTrue(((EquationToken *)[tokens objectAtIndex:0]).valid, NULL);
        STAssertTrue(((EquationToken *)[tokens objectAtIndex:1]).valid, NULL);
    }
    {
        Equation *equation = [[Equation alloc] initWithString:@"(())"];
        NSArray *tokens = equation.tokens;
        STAssertTrue(tokens.count == 4, NULL);
        STAssertTrue(((EquationToken *)[tokens objectAtIndex:0]).valid, NULL);
        STAssertTrue(((EquationToken *)[tokens objectAtIndex:1]).valid, NULL);
        STAssertTrue(((EquationToken *)[tokens objectAtIndex:2]).valid, NULL);
        STAssertTrue(((EquationToken *)[tokens objectAtIndex:3]).valid, NULL);
    }
    {
        Equation *equation = [[Equation alloc] initWithString:@"()()"];
        NSArray *tokens = equation.tokens;
        STAssertTrue(tokens.count == 4, NULL);
        STAssertTrue(((EquationToken *)[tokens objectAtIndex:0]).valid, NULL);
        STAssertTrue(((EquationToken *)[tokens objectAtIndex:1]).valid, NULL);
        STAssertTrue(((EquationToken *)[tokens objectAtIndex:2]).valid, NULL);
        STAssertTrue(((EquationToken *)[tokens objectAtIndex:3]).valid, NULL);
    }
    {
        Equation *equation = [[Equation alloc] initWithString:@")("];
        NSArray *tokens = equation.tokens;
        STAssertTrue(tokens.count == 2, NULL);
        STAssertFalse(((EquationToken *)[tokens objectAtIndex:0]).valid, NULL);
        STAssertFalse(((EquationToken *)[tokens objectAtIndex:1]).valid, NULL);
    }
    {
        Equation *equation = [[Equation alloc] initWithString:@"()()");
        NSArray *tokens = equation.tokens;
        STAssertTrue(tokens.count == 3, NULL);
        STAssertTrue(((EquationToken *)[tokens objectAtIndex:0]).valid, NULL);
        STAssertTrue(((EquationToken *)[tokens objectAtIndex:1]).valid, NULL);
        STAssertFalse(((EquationToken *)[tokens objectAtIndex:2]).valid, NULL);
    }
}
```

```

    Equation *equation = [[Equation alloc] initWithString:@"(())("];
    NSArray *tokens = equation.tokens;
    STAssertTrue(tokens.count == 6, NULL);
    STAssertTrue(((EquationToken *)[tokens objectAtIndex:0]).valid, NULL);
    STAssertTrue(((EquationToken *)[tokens objectAtIndex:1]).valid, NULL);
    STAssertTrue(((EquationToken *)[tokens objectAtIndex:2]).valid, NULL);
    STAssertTrue(((EquationToken *)[tokens objectAtIndex:3]).valid, NULL);
    STAssertTrue(((EquationToken *)[tokens objectAtIndex:4]).valid, NULL);
    STAssertFalse(((EquationToken *)[tokens objectAtIndex:5]).valid, NULL);
}
}

```

Testing Invalid Cases

You also should test for invalid cases. Listing 4–38 contains a `testInvalid:` method that tests for an array of invalid tokens and an invalid number (two decimal points).

Listing 4–38. *Testing Invalid Cases*

```

- (void)testInvalid
{
    Equation *equation = [[Equation alloc] initWithString:@"invalid0.3.3"];
    NSArray *tokens = equation.tokens;
    STAssertTrue(tokens.count == 8, NULL);
    [self helperTestToken:[tokens objectAtIndex:0] type:EquationTokenTypeInvalid
     value:@"i"];
    [self helperTestToken:[tokens objectAtIndex:1] type:EquationTokenTypeInvalid
     value:@"n"];
    [self helperTestToken:[tokens objectAtIndex:2] type:EquationTokenTypeInvalid
     value:@"v"];
    [self helperTestToken:[tokens objectAtIndex:3] type:EquationTokenTypeInvalid
     value:@"a"];
    [self helperTestToken:[tokens objectAtIndex:4] type:EquationTokenTypeInvalid
     value:@"l"];
    [self helperTestToken:[tokens objectAtIndex:5] type:EquationTokenTypeInvalid
     value:@"i"];
    [self helperTestToken:[tokens objectAtIndex:6] type:EquationTokenTypeInvalid
     value:@"d"];
    [self helperTestToken:[tokens objectAtIndex:7] type:EquationTokenTypeNumber
     value:@"0.3.3"];
    STAssertFalse(((EquationToken *)[tokens objectAtIndex:7]).valid, NULL);
}

```

Run all these tests to verify that they all pass. Congratulations—your tokenizing work is complete!

Showing the Equation

You’ve set the equation entry field to show rich text, and you’ve tokenized the equations, but you haven’t yet integrated the tokenizing into the equation entry field. You also haven’t done anything about syntax highlighting or pushing exponents to

superscript. In this section, you integrate the `tokenize:` routine into the equation entry controller, adding syntax highlighting and exponent superscripting while you're at it.

Many syntax highlighting applications allow users to configure the colors used for various bits of text, and some even allow users to create and share themes of colors that span all the various token types. We've opted not to be so accommodating, choosing the simpler route of hard-coding the colors. Hard-coding the colors directly into the `EquationToken` instances themselves is tempting, but `EquationToken`s really are model objects, and color is a view attribute, so instead we're going to put color under the control of the `EquationEntryViewController` object. This way, `EquationToken`s don't have any control over how they're displayed, and different views can display them differently with respect to colors or fonts.

Setting the Colors

You want to control both the foreground colors and the background colors for the tokens. For the foreground colors, create a dictionary called `COLORS` that uses the type of the token as the key and an `NSColor` instance as the value. For the background colors, use white for everything except invalid tokens, for which you use red. Start by implementing the foreground colors. Open `EquationEntryViewController.m` and import the `EquationToken.h` header:

```
#import "EquationToken.h"
```

Declare a static `NSDictionary` instance:

```
static NSDictionary *COLORS;
```

Fill the dictionary in the class's `initialize:` method. Because `NSDictionary` requires objects for keys and the type field of an `EquationToken` instance is an enum, which is a primitive `int`, convert the type to an `NSNumber` instance, which is an object, before using it as a key. Listing 4–39 shows the `initialize:` method.

Listing 4–39. *Filling the Colors Dictionary*

```
+ (void)initialize
{
    COLORS = [NSDictionary dictionaryWithObjectsAndKeys:
        [NSColor whiteColor], [NSNumber numberWithInt:EquationTokenTypeInvalid],
        [NSColor blackColor], [NSNumber numberWithInt:EquationTokenTypeNumber],
        [NSColor blueColor], [NSNumber numberWithInt:EquationTokenTypeVariable],
        [NSColor brownColor], [NSNumber numberWithInt:EquationTokenTypeOperator],
        [NSColor purpleColor], [NSNumber numberWithInt:EquationTokenTypeOpenParen],
        [NSColor purpleColor], [NSNumber numberWithInt:EquationTokenTypeCloseParen],
        [NSColor orangeColor], [NSNumber numberWithInt:EquationTokenTypeExponent],
        [NSColor cyanColor], [NSNumber numberWithInt:EquationTokenTypeSymbol],
        [NSColor magentaColor], [NSNumber numberWithInt:EquationTokenTypeTrigFunction],
        [NSColor whiteColor], [NSNumber numberWithInt:EquationTokenTypeWhitespace],
        nil];
}
```

We're somewhat arbitrary in our color selections, so feel free to change them. Notice, though, that the code uses `NSDictionary`'s `dictionaryWithObjectsAndKeys:` static method, which takes entries in value, key order. Notice also that the code converts the `EquationToken` types to objects using `NSNumber`'s `numberWithInt:` method.

Now it's time to colorize the equation strings.

Colorizing the Equation

Each time the user types in the equation entry field, the `controlTextDidChange:` method in `EquationEntryViewController` fires. Graphique currently uses that method to validate the equation. Now, you'll add code to that method to colorize the equation. Earlier in this chapter, we discussed Cocoa's attributed strings. At long last, we're ready to use them to add color attributes to the equation.

The existing `controlTextDidChange:` method creates an equation instance from the text in the equation text field. Now, it will create a mutable attributed string from the text in the equation text field, add color attributes to the mutable attributed string, and set the mutable attributed string back into the equation text field. When these steps complete, the equation entry will have syntax highlighting colors.

To add the color attributes, loop through all the tokens in the equation, using an index variable, `i`, to keep track of where the current token is in the attributed string. For each token, create a range (an `NSRange` instance) that corresponds to the range of characters that the current token occupies in the attributed string. Look up the proper foreground color for the token in the `COLORS` dictionary and set the foreground color using this code:

```
// Add the foreground color
[attributedString addAttribute:NSForegroundColorAttributeName value:[COLORS
objectForKey:[NSNumber numberWithInt:token.type]] range:range];
```

The `addAttribute:value:range` adds an attribute to an attributed string. The first parameter specifies the type of attribute to add, which in this case is `NSForegroundColorAttributeName`, meaning a foreground color attribute. The second parameter, `value`, specifies the value for the attribute, which in this case is the `NSColor` value corresponding to the current token's type. Finally, the `range` parameter specifies the range in the attributed string to which to apply the attribute you're adding.

Make a similar call to set the background color, only this time use `NSBackgroundColorAttributeName` for the first parameter. Determine the color by checking the valid member of token, passing a white color if the token is valid and a red color if the token is not valid. The call looks like this:

```
// Add the background color
[attributedString addAttribute:NSBackgroundColorAttributeName value:token.valid ?
[NSColor whiteColor] : [NSColor redColor] range:range];
```

Listing 4–40 shows the updated `controlTextDidChange:` method that colorizes the equation.

Listing 4–40. Colorizing the Equation

```

- (void)controlTextDidChange:(NSNotification *)notification
{
    Equation *equation = [[Equation alloc] initWithString: [self.textField stringValue]];

    // Create a mutable attributed string, initialized with the contents of the equation
    text field
    NSMutableAttributedString *attributedString = [[NSMutableAttributedString alloc]
initWithString:[self.textField stringValue]];

    // Variable to keep track of where we are in the attributed string
    int i = 0;

    // Loop through the tokens
    for (EquationToken *token in equation.tokens)
    {
        // The range makes any attributes we add apply to the current token only
        NSRange range = NSMakeRange(i, [token.value length]);

        // Add the foreground color
        [attributedString addAttribute:NSForegroundColorAttributeName value:[COLORS
objectForKey:[NSNumber numberWithInt:token.type]] range:range];

        // Add the background color
        [attributedString addAttribute:NSBackgroundColorAttributeName value:token.valid ?
[NSColor whiteColor] : [NSColor redColor] range:range];

        // Advance the index to the next token
        i += [token.value length];
    }
    // Set the attributed string back into the equation entry field
    [self.textField setAttributedStringValue:attributedString];

    NSError *error = nil;
    if (![equation validate:&error])
    {
        // Validation failed, display the error
        [feedback setStringValue:[NSString stringWithFormat:@"Error %d: %@", [error
code],[error localizedDescription]]];
    }
    else
    {
        [feedback setStringValue:@""];
    }
}

```

You haven't run the application in a while, but now is a good time. Build and run Graphique and start typing an equation. Experiment with both valid and invalid equations. You should see your equations in color, which is difficult to show with grayscale screenshots, but Figure 4–18 shows a sample equation with a parenthetical problem. You'll also notice an error message complaining that you've typed an invalid character—the letters in “sin.” You'll fix that later this chapter.



Figure 4-18. A syntax-colored equation

Superscripting Exponents

If you type an exponent in the equation entry field, however, whether implicit (follows a variable or a close parenthesis) or explicit (follows a caret), you'll notice that it isn't superscripted at all, as shown in Figure 4-19. All three instances of "2" in that equation are exponents, as evidenced by their orange foreground color. To make them appear in superscript, we must add attributes to our attributed string to superscript them.



Figure 4-19. Exponents that don't display as superscript

An attributed string supports an attribute called `NSSuperscriptAttributeName` that sounds tempting but in practice doesn't provide enough display control for our needs. Its documentation says its value parameter is an `NSNumber` containing an integer, but it doesn't tell you how far upward that integer will shift your superscripted attribute. Playing with this attribute shows that you want a little better control in how far your superscripted text is offset from the baseline. Fortunately, attributed strings offer an attribute type named `NSBaselineOffsetAttributeName`, whose value parameter is documented to take an `NSNumber` containing a floating-point value that represents the points to offset the text from the baseline.

Moving the text upward is insufficient, however; it should also shrink by setting an `NSFontAttributeName` attribute. You could play with absolute values for both the text size and the baseline offset, but you may want to change the size of the equation entry field's font (and in the next chapter, you indeed do). It's better, instead, to make the exponent text a percentage of the current text's size and move it off the baseline some percentage of the size as well. Because you can get the size of the current text, you can easily accomplish this. For exponents, then, make them half the height of the other text and move them halfway up the baseline.

The place to add the code to superscript the exponents is right after setting the background color but before advancing the index to the next token. This code should first determine whether the current token is an exponent and, if it is, perform the following actions in this sequence:

1. Calculate the height of the exponent and baseline shift by multiplying the text height by 0.5.
2. Set the exponent's font to a system font with the size calculated.
3. Shift the exponent's baseline upward by the size calculated.

Listing 4-41 shows the updated `controlTextDidChange:` method with the code to superscript the exponents.

Listing 4–41. Superscripting the Exponents

```

- (void)controlTextDidChange:(NSNotification *)notification
{
    Equation *equation = [[Equation alloc] initWithString: [self.textField stringValue]];

    // Create a mutable attributed string, initialized with the contents of the equation
    text field
    NSMutableAttributedString *attributedString = [[NSMutableAttributedString alloc]
initWithString:[self.textField stringValue]];

    // Variable to keep track of where we are in the attributed string
    int i = 0;

    // Loop through the tokens
    for (EquationToken *token in equation.tokens)
    {
        // The range makes any attributes we add apply to the current token only
        NSRange range = NSMakeRange(i, [token.value length]);

        // Add the foreground color
        [attributedString addAttribute:NSForegroundColorAttributeName value:[COLORS
objectForKey:[NSNumber numberWithInt:token.type]] range:range];

        // Add the background color
        [attributedString addAttribute:NSBackgroundColorAttributeName value:token.valid ?
[NSColor whiteColor] : [NSColor redColor] range:range];

        // If token is an exponent, make it superscript
        if (token.type == EquationTokenTypeExponent)
        {
            // Get the height of the rest of the text
            CGFloat height = [[textField font] pointSize] * 0.5;

            // Set the exponent font height
            [attributedString addAttribute:NSFontAttributeName value:[NSFont
systemFontOfSize:height] range:range];

            // Shift the exponent upwards
            [attributedString addAttribute:NSBaselineOffsetAttributeName value:[NSNumber
numberWithInt:height] range:range];
        }

        // Advance the index to the next token
        i += [token.value length];
    }
    // Set the attributed string back into the equation entry field
    [self.textField setAttributedStringValue:attributedString];

    NSError *error = nil;
    if (![equation validate:&error])
    {
        // Validation failed, display the error
        [feedback setValue:[NSString stringWithFormat:@"Error %d: %@", [error
code], [error localizedDescription]]];
    }
}

```



```

else
{
    [feedback setStringValue:@""];
}
}

```

Now run the application and type the same equation. Your exponents should be smaller and should shift upward, as shown in Figure 4–20.



Figure 4–20. *The exponents, smaller and shifted upward*

The equation entry field now stands complete, but you still must integrate it into the validator and tell the graphing evaluation how to interpret things like implicit multiplication. Keep reading to finish integrating the equation entry field into the application.

Updating the Validator

The equation entry field has grown from its humble beginnings, but the validator hasn't kept pace. For example, it flags the trigonometric functions as invalid, shown in Figure 4–21, even though you've explicitly added support for them. It's fooled by invalid parenthesis matching like `"()"` as well, shown in Figure 4–22, even though you've built a better parenthesis matcher. It's time for an upgrade.

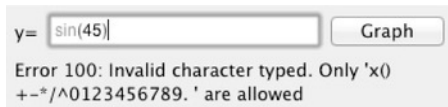


Figure 4–21. *The old validator flagging `sin()` as invalid*

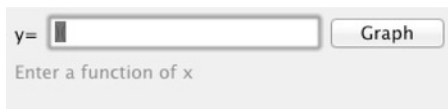


Figure 4–22. *The old validator fooled by invalid parentheses*

Happily, you can use the newly tokenized equation to do the validation. Remember that each token stores whether or not it's valid inside its `valid` member? You can loop through the tokens in the equation and test each token's `valid` member to enforce validation.

You've also expanded the list of acceptable characters to include trigonometric functions and the symbols `pi` and `e`, so you should update the message for invalid entry. Refer to the comment in the `validate:` method, which is in `Equation.m`, and update the requirements for the first rule:

```
// Validation rules
```

```
// 1. Only digits, operators, variables, parentheses, trigonometric functions, and
symbols allowed
// 2. There should be the same amount of closing and opening parentheses
// 3. no two consecutive operators
```

Gut the rest of the `validate:` method and add the code to loop through the equation's tokens and test each for validity. If we get an invalid token, test its type to see what error code to return: 102 for an invalid open parenthesis, 103 for an invalid close parenthesis, and 100 for all other invalid tokens.

To enforce rule #3, no two consecutive operators, store a pointer to the previous token in the loop, so you can test for back-to-back operators. Listing 4–42 shows the updated `validate:` method.

Listing 4–42. *The Updated `validate:` Method*

```
- (BOOL)validate:(NSError**)error
{
    // Validation rules
    // 1. Only digits, operators, variables, parentheses, trigonometric functions, and
symbols allowed
    // 2. There should be the same amount of closing and opening parentheses
    // 3. no two consecutive operators

    NSString *allowed = @"x, 0-9, (), operators, trig functions, pi, and e";
    EquationToken *previousToken = nil;
    for (EquationToken *token in self.tokens)
    {
        if (!token.valid)
        {
            if (token.type == EquationTokenTypeOpenParen)
            {
                return [self produceError:error withCode:102 andMessage:@"Too many open
parentheses"];
            }
            else if (token.type == EquationTokenTypeCloseParen)
            {
                return [self produceError:error withCode:103 andMessage:@"Too many closed
parentheses"];
            }
            else
            {
                return [self produceError:error withCode:100 andMessage:[NSString
stringWithFormat:@"Invalid character typed. Only %@ are allowed", allowed]];
            }
        }
        if (token.type == EquationTokenTypeOperator && previousToken.type ==
EquationTokenTypeOperator)
        {
            return [self produceError:error withCode:101 andMessage:@"Consecutive operators
are not allowed"];
        }
        previousToken = token;
    }
    return YES;
}
```

```
}
```

After updating the `validate:` method, run the Graphique tests and verify that they all still pass. They all should, because you've enforced the same rules and returned the same error codes for the same conditions. Now, run Graphique and enter some invalid input. You can see that the rules are all applied as you'd expect. See, for example, Figure 4-23, which shows some invalid input.

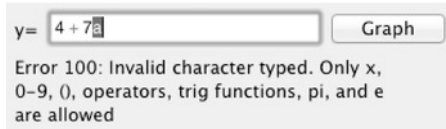


Figure 4-23. Invalid input validated by our new validator

Updating the Evaluator

You've expanded the range of what's considered valid input in the equation entry field. You've added implicit multiplication, implicit exponents, and the symbols `pi` and `e`. The graphing function still works as long as equations don't use these additions, but any equation that uses the additional functionality doesn't graph properly. Compare, for example, the graph for x^2 in Figure 4-24 with the graph for x^2 in Figure 4-25. They should be identical, but they obviously differ.

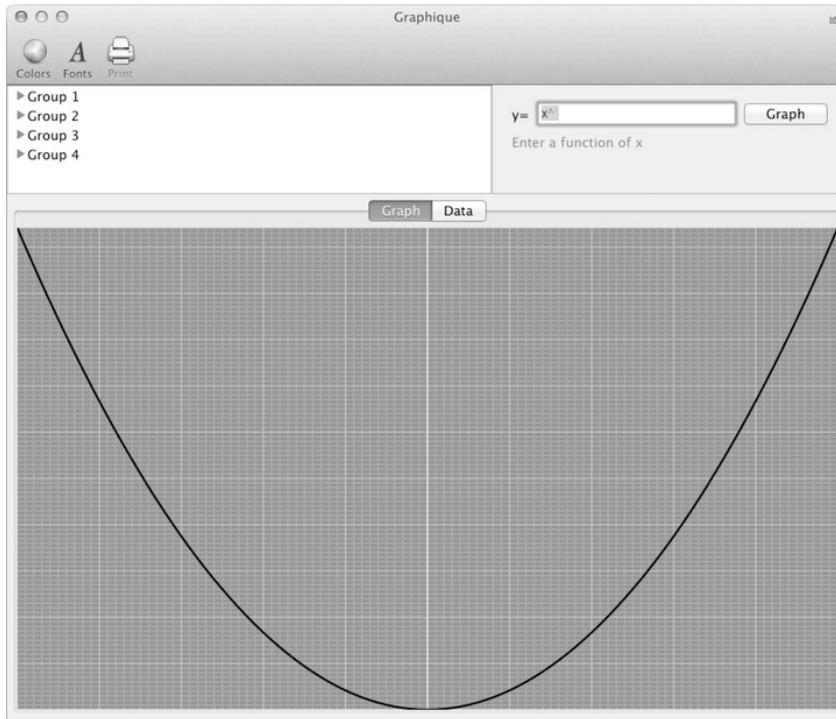


Figure 4-24. The graph for x^2

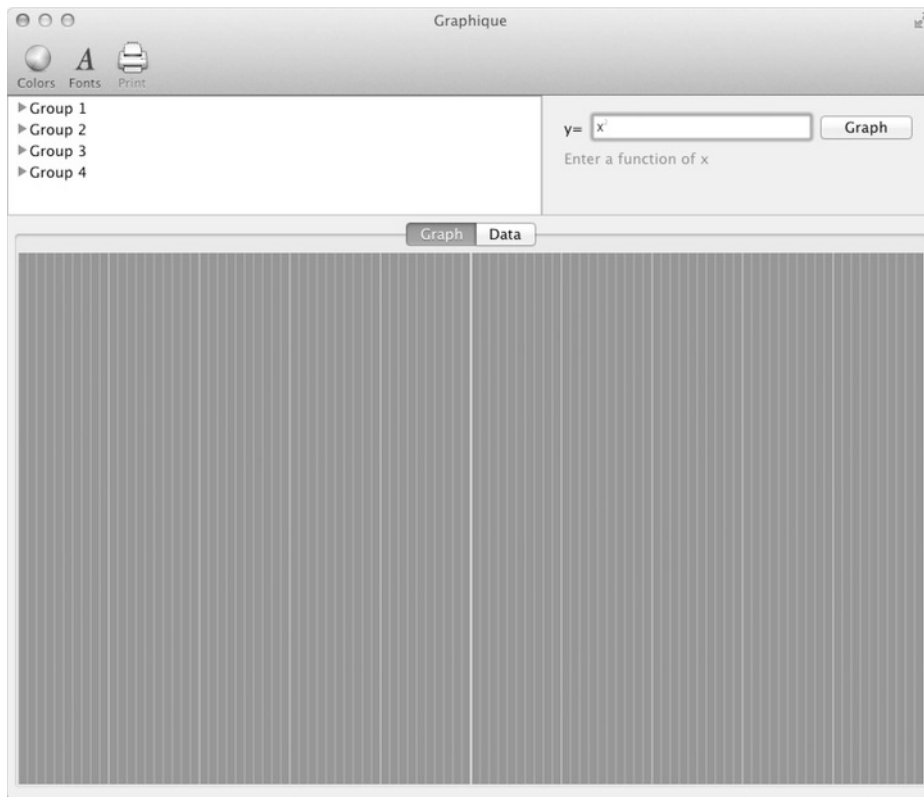


Figure 4–25. *The graph for x^2*

The graph for x^2 shows a nice parabola. The graph for x^2 remains starkly blank. Although both the parser and the validator recognize the 2 as an exponent, the evaluator doesn't.

As you'll recall, the evaluator uses `awk` to evaluate the equation, passing the text of the equation entry field to `awk` as the equation. To add support for implicit multiplication, implicit exponents, and symbols, you must alter the equation that we send to `awk`. Do that in a method called `expand:`. Declare this method in the class extension at the top of `Equation.m`, as shown in Listing 4–43.

Listing 4–43. *Declaring the `expand:` Method*

```
@interface Equation ()
- (BOOL)produceError:(NSError**)error withCode:(NSInteger)code
andMessage:(NSString*)message;
- (void)tokenize;
- (EquationToken *)newTokenFromString:(NSString *)string;
- (NSString *)expand;
@end
```

In the implementation of `expand:`, build an equation string by iterating through the equation's tokens. When you detect that a particular token requires special handling,

inject any special handling into the expanded string. When you're done iterating through the equation's tokens, return the expanded string.

The following are the special cases you must adjust for:

- For implicit exponents, if the current token is an exponent and the previous token isn't already a ^ operator, insert a ^ operator before appending the exponent.
- For implicit multiplication, if the current token is an open parenthesis and the previous token is a variable or a number, insert a * operator before appending the open parenthesis.
- Also for implicit multiplication, if the current token is a variable or a symbol and the previous token is a number, insert a * operator before appending the variable or symbol.
- For pi (whether "pi" or " π "), put in the value `M_PI` from `math.h` instead.
- For e, put in the value `M_E` from `math.h` instead.

See Listing 4-44 for the implementation of the `expand:` method.

Listing 4-44. *The `expand:` Method*

```
- (NSString *)expand
{
    NSMutableString *expanded = [NSMutableString string];

    EquationToken *previousToken = nil;
    for (EquationToken *token in self.tokens)
    {
        // Get the value of the current token
        NSString *value = token.value;

        if (previousToken != nil)
        {
            // Do implicit exponents
            if (token.type == EquationTokenTypeExponent && ![previousToken.value
isEqualToString:@"^"])
            {
                [expanded appendString:@"^"];
            }

            // Do implicit multiplication when token is an open parenthesis
            if (token.type == EquationTokenTypeOpenParen && (previousToken.type ==
EquationTokenTypeVariable || previousToken.type == EquationTokenTypeNumber))
            {
                [expanded appendString:@"*"];
            }

            // Do implicit multiplication when token is a variable or symbol
            if ((token.type == EquationTokenTypeVariable || token.type ==
EquationTokenTypeSymbol) && previousToken.type == EquationTokenTypeNumber)
            {

```

```

        [expanded appendString:@"*"];
    }
}

// Convert pi
if ([value isEqualToString:@"pi"] || [value isEqualToString:@"\u03c0"])
{
    value = [NSString stringWithFormat:@"%f", M_PI];
}

// Convert e
if ([value isEqualToString:@"e"])
{
    value = [NSString stringWithFormat:@"%f", M_E];
}

// Append the current token's value, which we may have adjusted
[expanded appendString:value];

// Keep a pointer to the previous token
previousToken = token;
}
return expanded;
}

```

You now must change your code in two places in `Equation.m` to call the `expand:` method:

- In `evaluateForX:`, where it builds the equation string to pass to `awk`
- In `description:`, where it returns the equation string it's graphing

Listing 4–45 shows the updated methods.

Listing 4–45. *The Updated `evaluateForX:` and `description:` Methods*

```

- (float)evaluateForX:(float)x
{
    NSTask *task = [[NSTask alloc] init];
    [task setLaunchPath: @"/usr/bin/awk"];

    NSArray *arguments = [NSArray arrayWithObjects: [NSString stringWithFormat:@"BEGIN {
x=%f ; print %@ ; }", x, [self expand]], nil];
    [task setArguments:arguments];

    NSPipe *pipe = [NSPipe pipe];
    [task setStandardOutput:pipe];

    NSFileHandle *file = [pipe fileHandleForReading];
    [task launch];

    NSData *data = [file readDataToEndOfFile];

    NSString *string = [[NSString alloc] initWithData:data encoding:
   :NSUTF8StringEncoding];
    float value = [string floatValue];
}

```

```

    return value;
}

- (NSString *)description
{
    return [NSString stringWithFormat:@"Equation [ %@]", [self expand]];
}

```

Now, run Graphique again, and try some equations with pi, e, implicit multiplication, and implicit exponents. Figure 4–26 shows the graph for the equation “ $2x^2 + 3(x + \pi)^3$.”

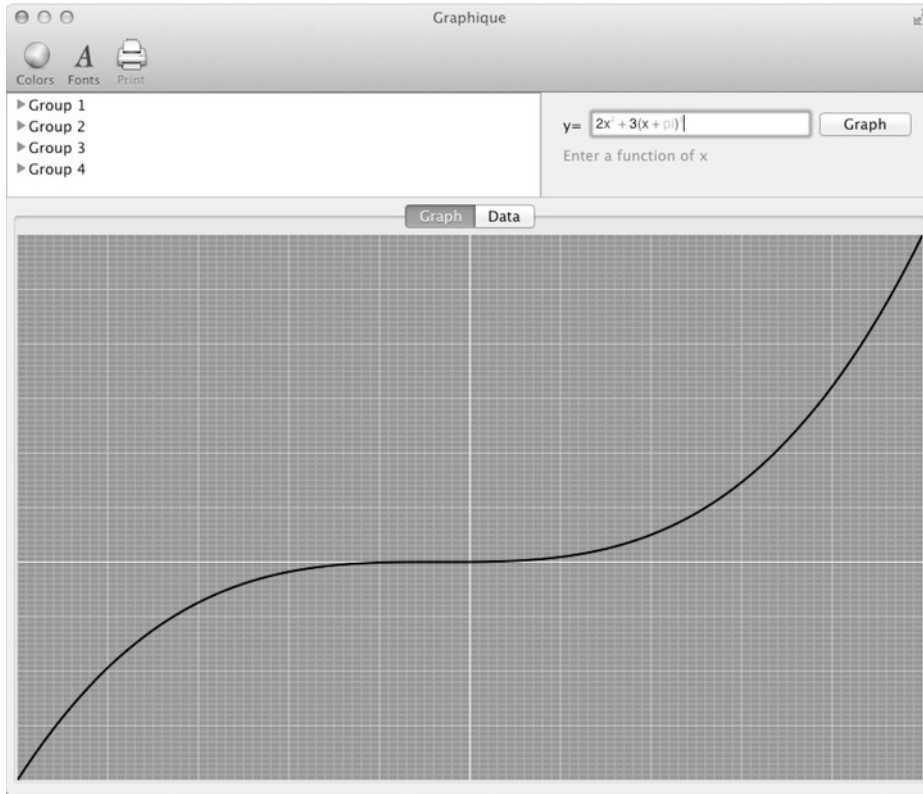


Figure 4–26. The updated equation validator with implicit exponents, implicit multiplication, and pi

Put a breakpoint on the first line of the `evaluateForX:` method and click the Graph button. Execution will stop at that line of code, and in the debugger window, type the following:

```
po self
```

and hit Enter. You’ll see the expanded equation, as shown here:

```
(gdb) po self
Equation [2*x^2 + 3*(x + 3.141593)^3]
```

Summary

Programmers often speak disdainfully about improved user interfaces, calling them “eye candy” and haughtily referring to themselves as “back-end programmers.” You can ride disdain for UI to obscurity, or you can focus on improving the UI for your programs so that they’re easier to use and they provide more value. They say a picture is worth 1,000 words, and we can see that a graph is worth more than 101 values in a table. You gain much more understanding about an equation from a graph than from a textual list of values.

Graphique’s improved equation editor makes equations easier to enter and understand as well. By colorizing the equations and matching their syntax to how people normally phrase equations, we minimize barriers for end users and invite them to use and experiment with Graphique.

User Preferences and the File System

Every time you launch Graphique, it starts over. Any equations you've entered disappear. Any graphs you've created, however clever or stunning, vanish. Graphique has no permanence. This might be fine for utility or one-shot applications, but users expect more out of applications like Graphique. They expect to be able to recall recent work or preserve output. They also expect to be able to set some preferences to customize behavior. In this chapter, we add some permanence to Graphique.

One of the things we do in this chapter is finally use the Fonts and Colors toolbar items to allow users to set the font used to enter equations and the color used to draw the line of the graph. We also implement the Preferences menu item so that users can decide whether the Graph view or the Data view appears first when Graphique launches.

We introduce you to writing data to the file system by allowing users to export their graphs as image files, so they can keep their graphs, post them to Flickr, or tweet them.

At the end of this chapter, Graphique will look like Figure 5–1. What's more, it will have permanence.

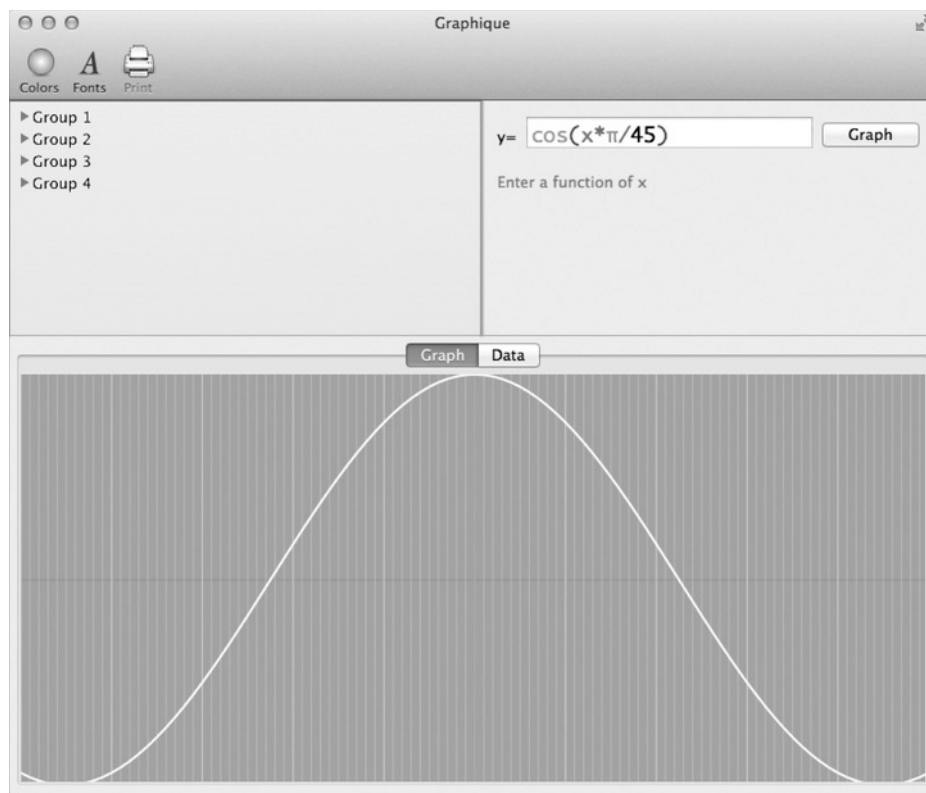


Figure 5–1. Graphique with permanence

Managing User Preferences

Major operating systems provide ways for applications to store user preferences, or settings, so that users can configure applications to look or behave a certain way and have them continue to look and behave that way any time they're launched. Microsoft Windows, for example, provides the system registry and an API on top of it to store user preferences. Linux and the various Unix flavors traditionally use dot files, which are hidden files whose names start with a dot (for example, `.vimrc`) that are stored in a user's home directory and contain the user preferences. Despite being a Unix-based operating system, OS X provides yet another way to store user preferences: user defaults, also known as *user preferences*, that are stored in property lists, which are binary XML files with a `.plist` extension and are often called *plist files*. They're usually found in a user's home directory, under the `Library/Preferences` directory, and use the applications' bundle identifiers and names in their file names. You can find one for Xcode, for example, called `com.apple.Xcode.plist`. Depending on whether you've clicked any toolbar buttons in Graphique, you may find that one for Graphique already exists. If it does, it's called `book.macdev.Graphique.plist`. You'll recognize `book.macdev.Graphique` as the bundle identifier you entered when you first created the project in Chapter 1.

Opening a property list file in most text editors reveals some recognizable strings like `NSObject` splattered among a load of gibberish, forcing you to admit that these files do indeed use a binary format. Apple provides an editor for these files as part of the Developer Tools called Property List Editor. Like Interface Builder, it was a stand-alone tool before Xcode 4's release but now comes integrated with Xcode. It displays a property list's values inside a two-column table. Other options for editing plist files exist as well: BBEdit and TextWrangler, two text editors offered by Bare Bones Software (<http://barebones.com>), decode the binary data from the file and display the file as if it were normal XML. Fat Cat Software offers a specialized property list editor called PlistEdit Pro (www.fatcatsoftware.com/plisteditpro/) that offers both a structured editor and a text editor. You can use Xcode's Property List Editor or one of the third-party offerings to edit these files. Take care when editing these files, however, because messing them up can cause some applications not to launch and can even cause problems with the operating system. It's no coincidence that Apple doesn't provide tools for nondevelopers to edit these files.

What Apple does provide to all users, however, is a command-line tool called `defaults`. This tool allows you to both read and write user defaults. To see the `defaults` tool work, go to a terminal prompt and type the following:

```
defaults read com.apple.dock
```

You'll see some output that reflects your preferences for the Dock that comes with OS X. The output on one of our machines, for example, starts like this:

```
{
    autohide = 1;
    "checked-for-launchpad" = 1;
    "mod-count" = 55;
    orientation = left;
    "persistent-apps" = (
    );
```

We can change preferences settings in the property list files by passing `write` instead of `read` as the first argument to `defaults` and specifying what we want to change. We could, for example, turn off `autohide` for our Dock by typing the following:

```
defaults write com.apple.dock autohide -boolean NO
```

This will change the value in the property list file, and to make it take effect, you must restart the Dock by typing this:

```
killall Dock
```

Preferences in OS X applications can be set through the `defaults` tool, but most applications offer GUI screens as well to read and write user preferences. In this chapter, we let users set their preferences for the font used for the equation entry editor. We use the toolbar button that Graphique already sports, **Fonts**, to control this setting. We use the **Colors** toolbar button to allow users to change the color for the line drawn in the graph. We also create a custom preference panel that will appear when users select **Graphique ► Preferences...** from the application menu that lets users set the initial tab (Graph or data) to display when Graphique is launched.

Understanding NSUserDefaults

Apple provides the defaults tool for command-line preference interaction and the `NSUserDefaults` class for programmatically interacting with user preferences. The `NSUserDefaults` class allows you to get and set user preferences much as you get and set values in a dictionary object, and it reduces the property list file to an implementation detail. You as a programmer don't have to know anything about property list files to use `NSUserDefaults` (although understanding them comes in handy when debugging issues with your code).

To use `NSUserDefaults`, you call the `standardUserDefaults:` class method, which returns the current user's `NSUserDefaults` instance, which represents that user's defaults database. You then call type-specific getter and setter methods on the `NSUserDefaults` instance to get and set preference values as you need them. For example, the following code sets the default (for some fictional application) for the key `TwitterName` to the string `@hoop33`, the default for the key `MaxTweets` to the number 75, and the default for the key `CheckAutomatically` to the boolean `YES`:

```
NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];
[userDefaults setObject:@"@hoop33" forKey:@"TwitterName"];
[userDefaults setInteger:75 forKey:@"MaxTweets"];
[userDefaults setBool:YES forKey:@"CheckAutomatically"];
```

You can read the defaults back using this code:

```
NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];
NSString *twitterName = [userDefaults stringForKey:@"TwitterName"];
NSInteger maxTweets = [userDefaults integerForKey:@"MaxTweets"];
BOOL checkAutomatically = [userDefaults boolForKey:@"CheckAutomatically"];
```

Persisting Default Values

`NSUserDefaults` uses a caching mechanism to reduce disk access, so you can read and write user preference values in your applications without worrying about impacting I/O performance. `NSUserDefaults` takes care of periodically flushing the cache to disk. We'll verify this later in this chapter when we set a user default and then use the defaults tool from the command line to read Graphique's property list file from disk to see a delay before the value appears. If you want to flush the cache yourself, call `NSUserDefaults`'s `synchronize:` method directly, like this:

```
NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];
[userDefaults synchronize];
```

Understanding Search Domains

Up to now, we've used the words *preferences* and *defaults* interchangeably without justifying ourselves. In the real world, *preferences* means settings that the user has specifically indicated that he or she wants, while *defaults* means sensible settings that the programmer has set up beforehand to be used in lieu of any user-set preferences. In

Apple-world, however, *preferences* and *defaults* mean the same thing, and indeed the class is called `NSUserDefaults`, not `NSUserDefaultsPreferences`. To understand this, you must understand what Apple calls *search domains*.

When you set up a user defaults object using code like this:

```
NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];
```

Cocoa sets up a search list in a specific order for accessing default values. Each item in the search list is called a *domain*, and when applications access preference values through the `userDefaults` object, Cocoa starts at the top of the search list, looking for a value that matches the key specified. When it finds a match, it returns the value. If it doesn't find a match, it returns `nil`. The domain order in the search list is as follows:

1. Command-line arguments, which are passed in the format `-key value`
2. The application's property list file
3. Global defaults
4. Language-specific defaults for any of the user's preferred language settings
5. Registered defaults, which are temporary defaults registered by the application

You'll rarely use command-line arguments for anything other than occasional debugging, because users launch their apps through LaunchPad, the Dock, the Finder, or a quick-launcher like Alfred, QuickSilver, or LaunchBar, so they don't as a rule pass any command-line arguments. We've covered the second search domain, the application's property list file, and will continue to do so in the chapter. Your application may use some global settings, and we're not going to go into language-specific settings. Registered defaults, however, are worth exploring a little deeper.

If the search has arrived at registered defaults, it means that no command-line default has been set, the user hasn't specified anything in the application property list, no global default has been set, and no language-specific setting has been specified. We're at the end of the line, but before the search drops off the cliff into `nil`, registered defaults allow us as programmers to apply sensible defaults for any properties we read. So, instead of writing code like this:

```
NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];
NSString *importantSetting = [userDefaults objectForKey:@"Important"];
if (importantSetting == nil)
{
    // Whoa -- nothing has been set for this important value!
    // Set it to a reasonable default so our application doesn't crash
    importantSetting = @"ReasonableDefault";
}
```

we can instead register a default and be guaranteed that we'll get that value returned if no other value trumps it earlier in the search chain, so that the code to get the value is simply this:

```
NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];
NSString *importantSetting = [userDefaults stringForKey:@"Important"];
```

To register defaults, you call the `registerDefaults:` method on your `NSUserDefaults` instance, passing a dictionary of keys and values. You normally do this in your application delegate's class method `initialize:`, which is called when your application delegate's class loads. This is an example of registering defaults:

```
+ (void)initialize
{
    NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];
    NSDictionary *appDefaults = [NSDictionary dictionaryWithObject:@"ReasonableDefault"
forKey:@"Important"];
    [userDefaults registerDefaults:appDefaults];
}
```

Note that any defaults you register aren't persisted to disk, in the application's property list file or elsewhere. They're a volatile fallback mechanism to provide sensible defaults, so should stay in your code, in your application delegate's `initialize:` method, as long as your application relies on sensible default values.

Resetting Defaults to Reasonable Values

`NSUserDefaults` provides a static method called `resetStandardUserDefaults:` that is often mistaken as a method to rid the application's property list file of any defaults the user has set. That's not at all what this method does. Instead, it flushes the defaults cache and unloads it so that a subsequent call to `standardUserDefaults:` reloads the cache with the default search order. This can be useful if you alter the domain search order for any reason and need to reset it, but since you probably won't, we don't cover that in this book.

What you really are aiming for when you want to reset any user settings to reasonable defaults is to delete any defaults they've set in the application's property list and allow searches to fall through to your registered defaults. To delete a default, call the `removeObjectForKey:` method on your `NSUserDefaults` instance, passing the name of the key you want to remove. To reset the "Important" default to use our registered default, for example, you'd use this code:

```
NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];
[userDefaults removeObjectForKey:@"Important"];
```

Now, searches for the `Important` key won't find a match in the application's property list domain and will fall through to the registered default.

Setting the Font for the Equation Entry Field

Armed with an understanding of `NSUserDefaults`, we're ready to allow users to set the font used in the equation entry field and have that setting persist through subsequent launches of Graphique. When we built the user interface for Graphique, we added a toolbar at the top of the Graphique window that includes a Fonts toolbar button. You may have tried clicking that button; if you haven't, click it now. When you click the Fonts toolbar button,

the Font panel displays, as shown in Figure 5–2. The fonts listed on your machine will differ, depending on what fonts you’ve installed, but the panel itself should match.



Figure 5–2. *The Fonts panel*

The menu for Graphique also includes a way to open the Fonts panel: **Format ► Font ► Show Fonts**. If you select that menu item, you should see the same Fonts panel you saw when you clicked the Fonts toolbar button. To understand how Graphique knows to display the Fonts panel, let’s start by dissecting the **Show Fonts** menu item. Select **MainMenu.xib** in Xcode to open it in Interface Builder, and then drill down into the **Format ► Font ► Show Fonts** menu item. Once you’ve selected that, open the Connections inspector. Your Xcode should resemble Figure 5–3.

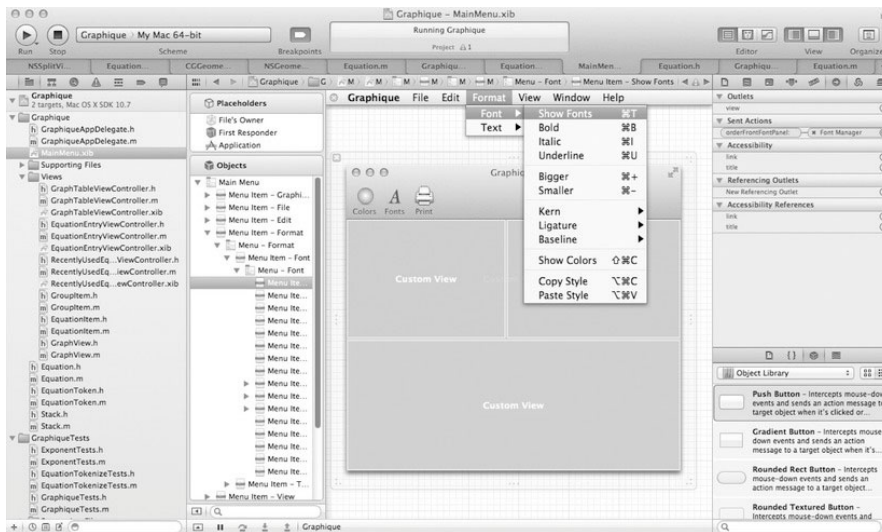


Figure 5–3. *Inspecting the connections of the Show Fonts menu item*

In the Sent Actions section, you see that the `orderFrontFontPanel:` selector is connected to Font Manager. Font Manager is an object that Xcode created for us when we created the Graphique project, and you can find it in the object hierarchy in `MainMenu.xib` (shown as the last item in Figure 5–4). It's of type `NSFontManager`, which you can verify in the Identity inspector, and it's connected to five menu items, as the Connections inspector (shown in Figure 5–5) shows: the Show Fonts menu item that we already looked at and the menu items for Bold, Italic, Smaller, and Bigger.

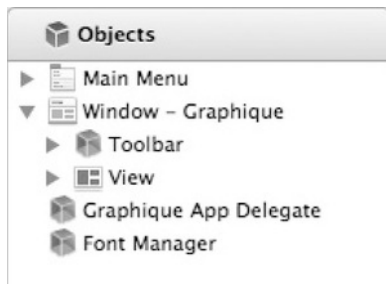


Figure 5–4. *The Font Manager object*

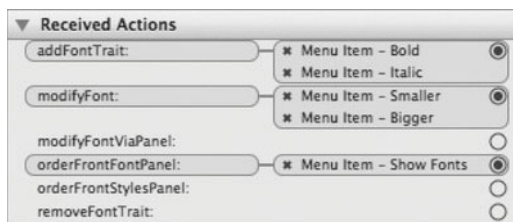


Figure 5–5. *The Font Manager's connections*

As you'd expect, the `orderFrontFontPanel:` selector is a method on `NSFontManager` that opens the Fonts panel, if it's not already open, and brings it to the front. You'd expect the Fonts toolbar item to be wired the same way, but if you select the Fonts toolbar item in the object hierarchy in Interface Builder and open the Connections inspector, you'll find no connections. Open the Attributes inspector instead, and you'll see that it lists its identifier as `NSToolbarShowFontsItem`, as shown in Figure 5–6.

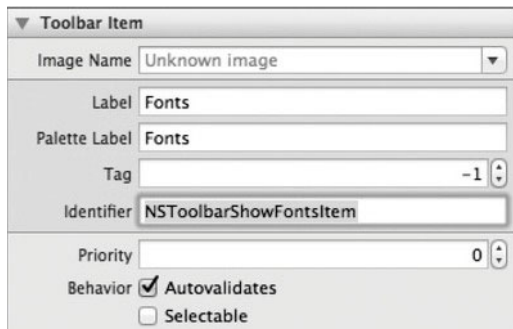


Figure 5–6. *The attributes for the Fonts toolbar item, including the `NSToolbarShowFontsItem` identifier*

When the toolbar detects that an item with this identifier is clicked, it knows to tell the Font Manager to open the Fonts panel. This is a standard toolbar item that comes with Cocoa.

Responding to Font Changes

After users display the Fonts panel, they can change the selected font. We must detect that change and update two things: the user preferences and the font used in the equation entry field. By updating the user preferences, we ensure that the user's selected font persists through application launches. By updating the font used in the equation entry field, we let the user see the chosen font.

When the selected font in the Fonts panel changes, the application's shared `NSFontManager` instance sends a message called `changeFont:` up what's called the responder chain. Think of the various objects that comprise your application as a line of eager customer service representatives ready to respond to your needs. When messages are sent to this line, each representative, or object, has the opportunity either to respond to the message or to ignore it and let the next object in line have the opportunity to respond. The message then continues to travel down the line until either someone responds or it passes the last object in line, typically your application delegate. This is a fairly typical pattern in event-oriented programming.

The entire responder chain doesn't always get the opportunity to respond to each message, because the message doesn't always start at the "front" of the line. Where the message starts depends on what currently has the focus in the application. The selected object is called the First Responder (which, like an unruly customer service representative, it can refuse to become, but let's not get too carried away with the metaphor), and the message starts there and works its way up the hierarchy. Understanding this is crucial to implementing our font-changing code. If, for example, we put our code to handle the font change in the equation entry controller, our code would get the `changeFont:` message only if the equation entry controller, or one of the objects it contains (such as the equation entry text field), has the focus. If the focus is currently anywhere else in the application, users could change the font in the Fonts panel and nothing would happen, which would lead to frustration and bug reports. Instead, we want the equation font to change in response to Fonts panel selection changes, no matter where the focus is in the application. We'll put our code, then, in our application delegate, so we'll always get the message.

We have one more hurdle to jump, though. As we said before, when the user changes the selected font in the Fonts panel, the `changeFont:` message is sent. We can put code to handle that message in our application delegate, and it would look something like this:

```
- (void)changeFont:(id)sender
{
    // Handle the font change
}
```

This works as long as the equation entry field doesn't have the focus. If the user is typing in the equation entry field, however, and then changes the font in the Fonts panel, nothing happens. Our `changeFont:` selector doesn't get called, and the font in the equation entry field remains the same. When the equation entry field has the focus, you see, it becomes the First Responder, and it swallows the `changeFont:` message and doesn't pass it on for our application delegate's implementation to handle. It's a little confusing why this happens, because it's an instance of `NSTextField`, as you'll recall, which has no `changeFont:` selector in its object hierarchy. When the equation entry field has the focus, however, Cocoa automatically overlays it with what it calls a *field editor* to handle the text entry and editing chores. This field editor is an `NSTextView` instance, which inherits from `NSText`, which implements `changeFont:` and swallows the message.

To help us avoid this pitfall, `NSFontManager` allows us to change the selector it calls when the font changes to anything of our choosing, so we can prevent the field editor from stealing our font change messages and can handle all font changes in our application delegate. We'll take advantage of this ability in the `initialize:` method of our application delegate after we set up the user defaults. Add this code in Listing 5-1 to `GraphiqueAppDelegate.m` to get the user defaults, register a reasonable font, and change the selector called when the font selection changes to `changeEquationFont:`.

Listing 5-1. Registering a Font in the User Defaults and Changing the Font Change Selector

```
+ (void)initialize
{
    // Get the user defaults
    NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

    // Set the font to a reasonable choice and convert to an NSData object
    NSFont *equationFont = [NSFont systemFontOfSize:18.0];
    NSData *fontData = [NSArchiver archivedDataWithRootObject:equationFont];

    // Set the font in the defaults
    NSDictionary *appDefaults = [NSDictionary dictionaryWithObject:fontData
                                                                    forKey:@"equationFont"];
    [userDefaults registerDefaults:appDefaults];

    // Change the action for the Font Panel so that the text field doesn't swallow the
    // changes
    [[NSFontManager sharedFontManager] setAction:@selector(changeEquationFont:)];
}
```

You'll notice that we had to convert the `NSFont` to an `NSData` object, because `NSUserDefaults` doesn't support `NSFont` instances, but it does support `NSData` instances.

Implementing `changeEquationFont:`

The next step is to implement the `changeEquationFont:` method to get the user defaults, pull out its existing equation entry font, and ask the `NSFontManager` (stored in the sender object passed to `changeEquationFont:`) to convert it to the new selected font. Then, the implementation should store the new font in the user defaults and tell the equation entry controller to update itself to use the new font. You'll find this code in Listing 5-2.

Listing 5–2. Responding to Font Selection Changes

```

- (void)changeEquationFont:(id)sender
{
    // Get the user defaults
   NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

    // Get the user's font selection and convert from NSData to NSFont
    NSData *fontData = [userDefaults dataForKey:@"equationFont"];
    NSFont *equationFont = (NSFont *)[NSUnarchiver unarchiveObjectWithData:fontData];

    // Convert the font to the new selection
    NSFont *newFont = [sender convertFont:equationFont];

    // Convert the new font into an NSData object and set it back into the user defaults
    fontData = [NSArchiver archivedDataWithRootObject:newFont];
    [userDefaults setObject:fontData forKey:@"equationFont"];

    // Tell the equation entry field to update to the new font
    [self.equationEntryViewController controlTextDidChange:nil];
}

```

Applying the New Font

The final step for responding to font changes is to actually use the new font in the equation entry field. In the `controlTextDidChange:` method in `EquationEntryViewController.m`, we already decorate the text with colors and exponent sizing. We'll augment this method to get the selected font from the user defaults and add a font attribute with the new font to the entire string. We also use the font manager's `setSelectedFont:isMultiple:` method to make sure that the font selected in the Fonts panel is the same font we're using in the equation entry field. That code looks like this:

```

// Get the user defaults
NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

// Get the selected font
NSData *fontData = [userDefaults dataForKey:@"equationFont"];
NSFont *equationFont = (NSFont *)[NSUnarchiver unarchiveObjectWithData:fontData];

// Set the selected font in the font panel
[[NSFontManager sharedFontManager] setSelectedFont:equationFont isMultiple:NO];

// Set the font for the equation to the selected font
[attributedString addAttribute:NSFontAttributeName value:equationFont
range:NSMakeRange(0, [attributedString length])];

```

We also change the code for the exponents to use the selected font. It looks like this:

```

// Calculate the height of the exponent as half the height of the selected font
CGFloat height = [equationFont pointSize] * 0.5;

// Set the exponent font height
[attributedString addAttribute:NSFontAttributeName value:[NSFont
fontWithName:equationFont.fontName size:height] range:range];

```

Finally, we adjust the height of the text field to fit the selected font, like this:

```
// Adjust the height of the equation entry text field to fit the new font size
NSSize size = [textField frame].size;
size.height = ceilf([equationFont ascender]) - floorf([equationFont descender]) + 4.0;
[textField setFrameSize:size];
```

The method now looks like Listing 5–3.

Listing 5–3. *The controlTextChanged: Method*

```
-(void)controlTextChanged:(NSNotification *)notification
{
    Equation *equation = [[Equation alloc] initWithString: [self.textField stringValue]];

    // Create a mutable attributed string, initialized with the contents of the equation
    text field
    NSMutableAttributedString *attributedString = [[NSMutableAttributedString alloc]
initWithString:[self.textField stringValue]];

    // Get the user defaults
    NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

    // Get the selected font
    NSData *fontData = [userDefaults dataForKey:@"equationFont"];
    NSFont *equationFont = (NSFont *)[NSUnarchiver unarchiveObjectWithData:fontData];

    // Set the selected font in the font panel
    [[NSFontManager sharedFontManager] setSelectedFont:equationFont isMultiple:NO];

    // Set the font for the equation to the selected font
    [attributedString addAttribute:NSFontAttributeName value:equationFont
range:NSMakeRange(0, [attributedString length])];

    // Variable to keep track of where we are in the attributed string
    int i = 0;

    // Loop through the tokens
    for (EquationToken *token in equation.tokens)
    {
        // The range makes any attributes we add apply to the current token only
        NSRange range = NSMakeRange(i, [token.value length]);

        // Add the foreground color
        [attributedString addAttribute:NSForegroundColorAttributeName value:[COLORS
objectForKey:[NSNumber numberWithInt:token.type]] range:range];

        // Add the background color
        [attributedString addAttribute:NSBackgroundColorAttributeName value:token.valid ?
[NSColor whiteColor] : [NSColor redColor] range:range];

        // If token is an exponent, make it superscript
        if (token.type == EquationTokenTypeExponent)
        {
            // Calculate the height of the exponent as half the height of the selected font
            CGFloat height = [equationFont pointSize] * 0.5;
```

```

        // Set the exponent font height
        [attributedString addAttribute:NSFontAttributeName value:[NSFont
fontWithName:equationFont.fontName size:height] range:range];

        // Shift the exponent upwards
        [attributedString addAttribute:NSBaselineOffsetAttributeName value:[NSNumber
numberWithInt:height] range:range];
    }

    // Advance the index to the next token
    i += [token.value length];
}

// Adjust the height of the equation entry text field to fit the new font size
NSSize size = [textField frame].size;
size.height = ceilf([equationFont ascender]) - floorf([equationFont descender]) + 4.0;
[textField setFrameSize:size];

// Set the attributed string back into the equation entry field
[self.textField setAttributedStringValue:attributedString];

// Clean up
[attributedString release];

NSError *error = nil;
if(![equation validate:&error])
{
    // Validation failed, display the error
    [feedback setStringValue:[NSString stringWithFormat:@"Error %d: %@", [error
code],[error localizedDescription]]];
}
else
{
    [feedback setStringValue:@""];
}
[equation release];
}

```

Build and run Graphique, bring up the Fonts panel, and change the font. You can see that, wherever your focus, you can change the selected font and see the new font in the equation entry panel. Figure 5–7, for example, shows the equation entry field using the Marker Felt font.

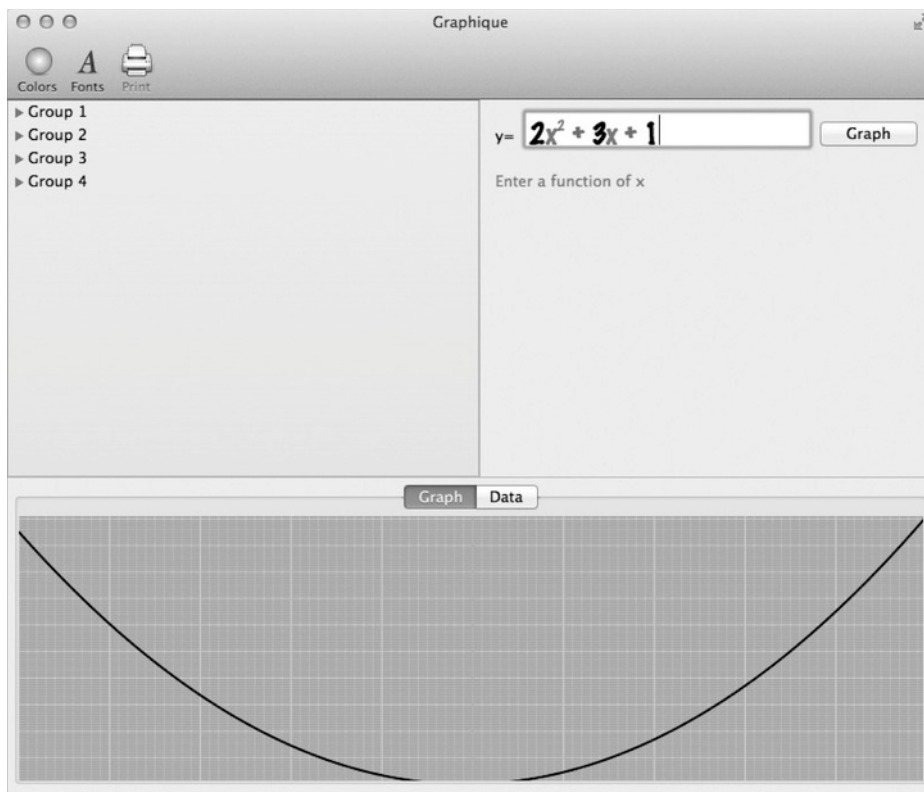


Figure 5–7. *The equation entry field using the Marker Felt font*

Setting the Line Color

Clicking the Colors button in Graphique's toolbar displays the color selection panel, as shown in Figure 5–8. Programming the color selection panel resembles programming the font selection panel, so you'll be able to transfer many of the concepts you just learned to selecting colors.

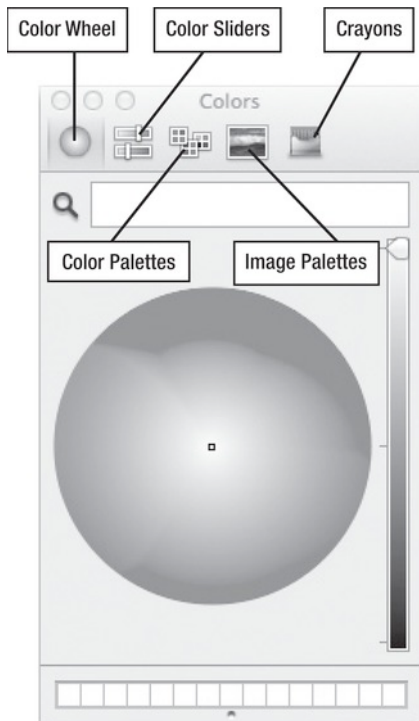


Figure 5–8. *The color selection panel*

Understanding Color Selection Modes

You'll notice in Figure 5–8 the five icons across the top of the color selection panel. Each of those icons represents a different color selection mode. The standard modes are as follows:

- Color Wheel
- Color Sliders
- Color Palettes
- Image Palettes
- Crayons

Each mode represents a different interface to select the same thing: a color. Although all five modes are available by default, you can restrict the Colors panel to display only some of these, and you can also create your own custom color selection modes. In Graphique, we demonstrate mode restriction by restricting the color selection to Crayons mode. To restrict the mode, you call `NSColorPanel`'s static `setPickerMask:` method, passing one or more mode masks bitwise OR'ed together. To restrict the Colors panel to Color Wheel and Color Palettes modes, for example, you'd code the following:

```
[NSColorPanel setPickerMask:
    (NSColorPanelWheelModeMask | NSColorPanelColorListModeMask)];
```

Note that you must make this call before any Colors panel instances have been created. If you display a Colors panel after calling the preceding code, it looks like Figure 5–9.

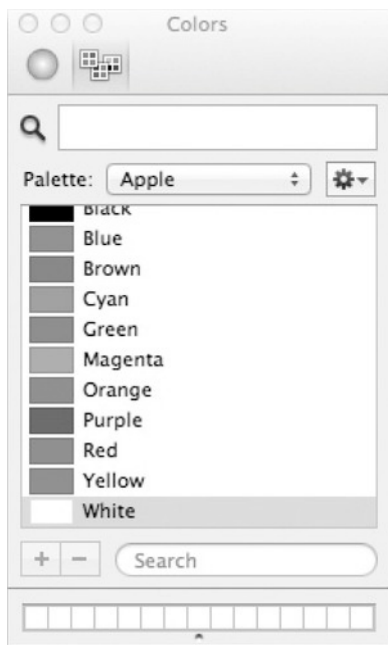


Figure 5–9. A Colors panel with only two modes

Displaying the Color Selection Panel

Clicking the Colors toolbar item passes the identifier `NSToolbarShowColorsItem` to the toolbar, which you can see in the Attributes inspector. This is similar to how the Fonts toolbar item passes the `NSToolbarShowFontsItem` identifier. When this identifier is passed, Cocoa knows to display the color selection panel.

For our implementation, we start by registering a reasonable user default for the line color: black. We also set the Colors panel to Crayons-only mode. Update the `initialize:` method in `GraphiqueAppDelegate.m` accordingly, as shown in Listing 5–4.

Listing 5–4. The updated `initialize:` Method

```
+ (void)initialize
{
    // Get the user defaults
    NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

    // Set the font to a reasonable choice and convert to an NSData object
    NSFont *equationFont = [NSFont systemFontOfSize:18.0];
    NSData *fontData = [NSArchiver archivedDataWithRootObject:equationFont];

    // Set the color to a reasonable choice and convert to an NSData object
```



```

NSColor *lineColor = [NSColor blackColor];
NSData *colorData = [NSArchiver archivedDataWithRootObject:lineColor];

// Set the font and color in the defaults
NSDictionary *appDefaults = [NSDictionary dictionaryWithObjectsAndKeys:fontData,
@"equationFont", colorData, @"lineColor", nil];
[userDefaults registerDefaults:appDefaults];

// Change the action for the Font Panel so that the text field doesn't swallow the
changes
[[NSFontManager sharedFontManager] setAction:@selector(changeEquationFont:));

// Set the Colors panel to show only Crayons mode
[NSColorPanel setPickerMask:NSColorPanelCrayonModeMask];
}

```

You'll notice that, as with the font, we had to convert the `NSColor` object to an `NSData` object before storing in the user defaults. You'll also notice that the key we use for the color is `lineColor`.

Responding to Color Changes

When users click a new color in the color selection panel, a `changeColor:` message is sent up the responder chain, similar to how the font selection panel sends a `changeFont:` message. As with the `changeFont:` message, however, the equation entry field swallows the `changeColor:` message when it has focus. Since we want to have the color selection always refer to the graph's line, we must change the message sent when the color changes so we can always catch it in our application delegate.

We must deal with one more twist, however—we must call `NSColorPanel`'s `setTarget:` method to set the target for our color change messages to be the application delegate. This means that, unlike with changing the font change messages, we can't make this change in the `initialize:` method. The `initialize:` method is a class method, called before our `GraphiqueAppDelegate` instance is created. We want our `GraphiqueAppDelegate` instance to be the target for our color change messages, because we have to tell the graph instance to redraw itself to the new color. Update the `applicationDidFinishLaunching:` method, adding code to set the target for color changes to the application delegate instance, and to set the action for when the color selection changes to `changeGraphLineColor:.` Stick these two lines of code at the end of that method:

```

[[NSColorPanel sharedColorPanel] setTarget:self];
[[NSColorPanel sharedColorPanel] setAction:@selector(changeGraphLineColor:)];

```

Add a declaration for the `changeGraphLineColor:` method to `GraphiqueAppDelegate.h`:

```
- (void)changeGraphLineColor:(id)sender;
```

We'll catch this message in our application delegate, store the new color in the user defaults, and then tell the graph view to redraw itself. Add a `changeGraphLineColor:` method to `GraphiqueAppDelegate.m` that matches Listing 5-5.

Listing 5–5. *The changeGraphLineColor: Method*

```

- (void)changeGraphLineColor:(id)sender
{
    // Set the selected color in the user defaults
    NSData *colorData = [NSArchiver archivedDataWithRootObject:[(NSColorPanel *)sender
color]];
    [[NSUserDefaults standardUserDefaults] setObject:colorData forKey:@"lineColor"];

    // Tell the graph to redraw itself
    [self.graphTableViewController.graphView setNeedsDisplay:YES];
}

```

NOTE: As of Mac OS X Lion and the use of Automatic Reference Counting (ARC), you can no longer have forward declarations in your code. Be sure to import `GraphView.h` at the top of `GraphiqueAppDelegate.m` so the compiler knows what class we are talking about.

The last thing we must do is update the `drawRect:` method in `GraphView.m` to use the selected color. The existing code, which looks like this, sets the line color (stored in the variable `curveColor`) to black:

```

// Set the color scheme
NSColor *background = [NSColor colorWithDeviceRed:0.30 green:0.58 blue:1.0 alpha:1.0];
NSColor *axisColor = [NSColor colorWithDeviceRed:1.0 green:1.0 blue:0.0 alpha:1.0];
NSColor *gridColorLight = [NSColor colorWithDeviceRed:1.0 green:1.0 blue:1.0 alpha:0.5];
NSColor *gridColorLighter = [NSColor colorWithDeviceRed:1.0 green:1.0 blue:1.0
alpha:0.25];
NSColor *curveColor = [NSColor colorWithDeviceRed:.0 green:0.0 blue:0 alpha:1.0];

```

Change the code to instead read the color from the user defaults, like this:

```

// Set the color scheme
NSColor *background = [NSColor colorWithDeviceRed:0.30 green:0.58 blue:1.0 alpha:1.0];
NSColor *axisColor = [NSColor colorWithDeviceRed:1.0 green:1.0 blue:0.0 alpha:1.0];
NSColor *gridColorLight = [NSColor colorWithDeviceRed:1.0 green:1.0 blue:1.0 alpha:0.5];
NSColor *gridColorLighter = [NSColor colorWithDeviceRed:1.0 green:1.0 blue:1.0
alpha:0.25];

// Get the line color from the user defaults
NSData *colorData = [[NSUserDefaults standardUserDefaults] dataForKey:@"lineColor"];
NSColor *curveColor = (NSColor *)[NSUnarchiver unarchiveObjectWithData:colorData];

```

Now you can build and run *Graphique* and, with a graph showing, open the color selection panel and select a new color. You'll see the line color immediately update to the new selection. Figure 5–10 shows the graph with the Snow crayon selected.

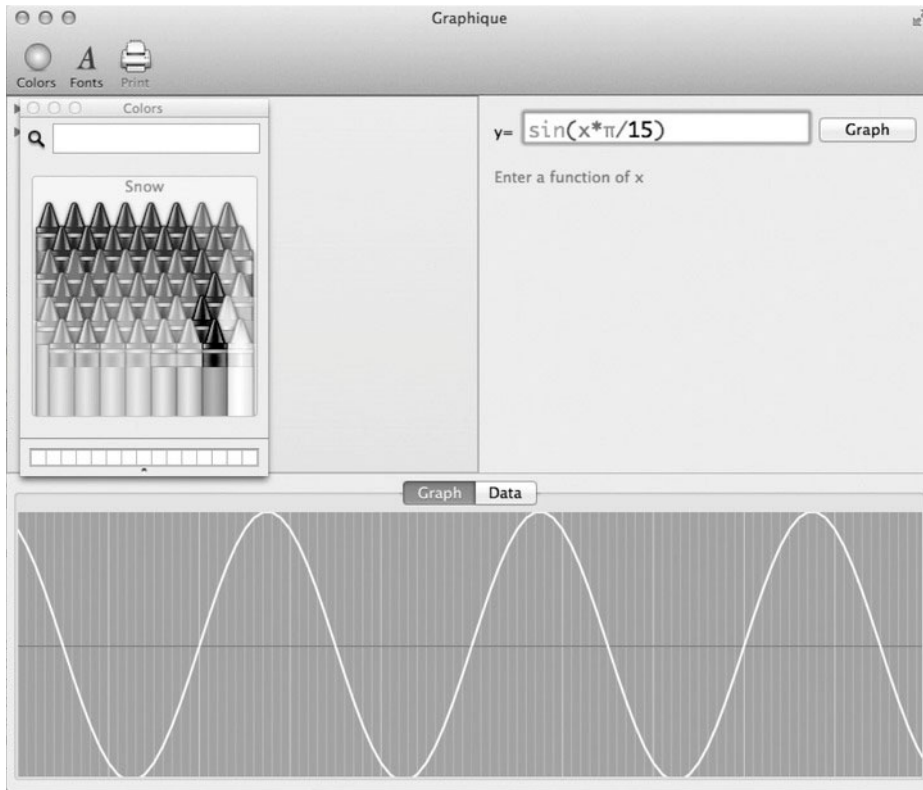


Figure 5–10. *The graph with the Snow crayon selected*

Creating a Custom Preferences Panel

Most OS X applications provide a menu item called Preferences that, when selected, shows a panel that allows users to set custom preferences. Graphique has a menu item called Preferences, but it's grayed out since we've provided no such preferences panel. In this section, we create a preference panel that displays when users select the Preferences menu item. It contains a single check box to determine whether the initial view for a rendered equation is the table view or the Graph view, and it looks like Figure 5–11.

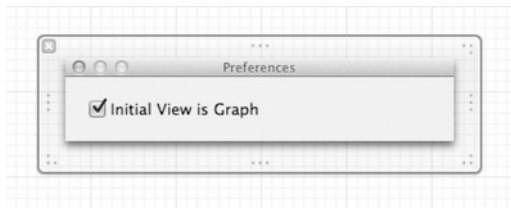


Figure 5–11. *The Preferences panel*

Creating the Preferences View

We'll start by creating the code to control the preferences view. Create a new Objective-C class in your Views group, make it a subclass of `NSWindowController`, and call it `PreferencesController`. In this class, we want to create an outlet for a check box that determines whether the initial view is the graph and create an action that responds when the user checks or unchecks that check box. The header file, `PreferencesController.h`, is shown in Listing 5-6.

Listing 5-6. *PreferencesController.h*

```
#import <Cocoa/Cocoa.h>

@interface PreferencesController : NSWindowController
{
    NSButton *initialViewIsGraph;
}

@property (nonatomic, retain) IBOutlet NSButton *initialViewIsGraph;

-(IBAction)changeInitialView:(id)sender;

@end
```

The implementation file for the `PreferencesController` does three things:

1. In the `init:` method, it loads `PreferencesController.xib`.
2. When the Preferences panel appears, it retrieves the value for the initial view from the user defaults and updates the check box accordingly.
3. When the user checks or unchecks the check box, it determines the state of the check box and updates the user defaults.

The code for `PreferencesController.m`, shown in Listing 5-7, implements the behavior for step 2 in the `windowDidLoad:` method, which is called when the Preferences window loads. The action method `changeInitialView:` implements the behavior for step 3.

Listing 5-7. *PreferencesController.m*

```
#import "PreferencesController.h"

@implementation PreferencesController

@synthesize initialViewIsGraph;

- (id)init
{
    self = [super initWithWindowNibName:@"PreferencesController"];
    return self;
}

- (void)windowDidLoad
{
    [super windowDidLoad];
}
```

```

// Get the user defaults
NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

// Set the checkbox to reflect the user defaults
[initialViewIsGraph setState:[userDefaults boolForKey:@"InitialViewIsGraph"]];
}

- (IBAction)changeInitialView:(id)sender
{
// Get the user defaults
NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

// Set the user defaults value for the initial view
[userDefaults setBool:[initialViewIsGraph state] forKey:@"InitialViewIsGraph"];
}

@end

```

Now we're ready to create the actual user interface with the check box.

Create a new Empty Interface Builder document, as shown in Figure 5–12, and call it `PreferencesController.xib`. Open `PreferencesController.xib` in Xcode and drag a Panel object onto the blank canvas. Next, drag a Check Box object onto the Panel object and change its text to “Initial View is Graph.” Resize the Panel object to get rid of most of the empty space. Open the Attributes inspector and change the Window Title to Preferences and uncheck the Resize box. Your view should look like Figure 5–13.

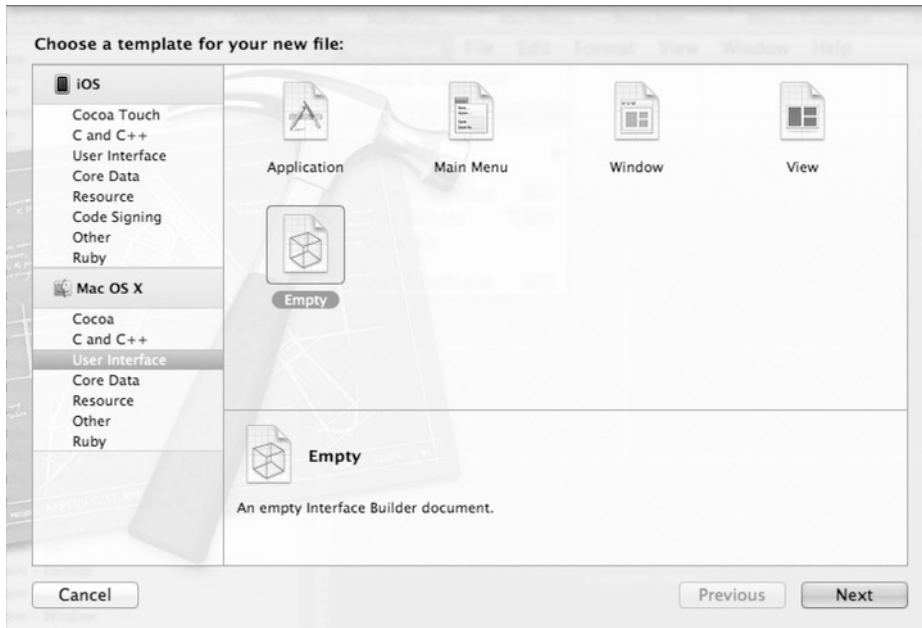


Figure 5–12. Selecting an Empty Interface Builder document

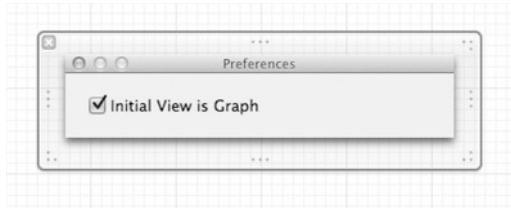


Figure 5-13. *The view for PreferencesController*

With PreferencesController.xib still open, select File's Owner, open the Identity inspector, and select PreferencesController as the class, as shown in Figure 5-14.

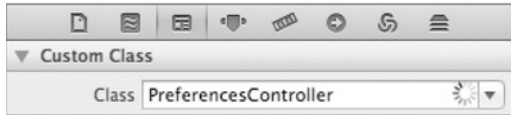


Figure 5-14. *File's Owner set to PreferencesController*

Now you can Ctrl+drag from the check box to File's Owner, select the `changeInitialView:` action, and then Ctrl+drag from File's Owner to the check box and select the `initialViewIsGraph` outlet. Finally, Ctrl+drag from File's Owner to the panel itself and select the window outlet. If you forget this step, the Preferences panel will display the first time you select the Preferences menu item (when we get that far), but once you've closed it, it won't display anymore, no matter how often you select the Preferences menu item.

We've now wired our user interface to the PreferencesController class, but we haven't yet done anything to make the PreferencesController user interface display. Read on to close that loop.

Displaying the Preferences Panel

To display the preferences panel we've created, we must tell our application delegate about it and wire it to the Preferences menu item in MainMenu.xib. Open GraphiqueAppDelegate.h and add a forward declaration for the PreferencesController class, a PreferencesController member to the interface, and a PreferencesController property. Also, add an action method to display the preferences panel. See Listing 5-8.

Listing 5-8. *Updating GraphiqueAppDelegate.h for PreferencesController*

```
#import <Cocoa/Cocoa.h>
#import <CoreData/CoreData.h>

@class EquationEntryViewController;
@class GraphTableViewController;
@class RecentlyUsedEquationsViewController;
@class PreferencesController;

@interface GraphiqueAppDelegate : NSObject <NSApplicationDelegate>

@property (strong) IBOutlet NSWindow *window;
```

```

@property (weak) IBOutlet NSSplitView *horizontalSplitView;
@property (weak) IBOutlet NSSplitView *verticalSplitView;
@property (strong) EquationEntryViewController *equationEntryViewController;
@property (strong) GraphTableViewController *graphTableViewController;
@property (strong) RecentlyUsedEquationsViewController
*recentlyUsedEquationsViewController;
@property (strong) PreferencesController *preferencesController;

- (void)changeGraphLineColor:(id)sender;
-(IBAction)showPreferencesPanel:(id)sender;

@end

```

In `GraphiqueAppDelegate.m`, import `PreferencesController.h`, add a `@synthesize` line for the `preferencesController` instance, and then implement the `showPreferencesPanel:` method. This method should instantiate the `preferencesController` instance, if it hasn't already been instantiated, and then display the preferences panel. See the code in Listing 5–9.

Listing 5–9. The Method to Show the Preferences Panel

```

- (IBAction)showPreferencesPanel:(id)sender
{
    // Create the preferences panel if we haven't already
    if (preferencesController == nil)
    {
        preferencesController = [[PreferencesController alloc] init];
    }

    // Show the panel
    [preferencesController showWindow:self];
}

```

All that's left to display the preferences panel is to wire it to the Preferences menu item. Open `MainMenu.xib`, select the Preferences menu item, Ctrl+drag to the Graphique App Delegate instance, and select `showPreferencesPanel:`, as shown in Figure 5–15.

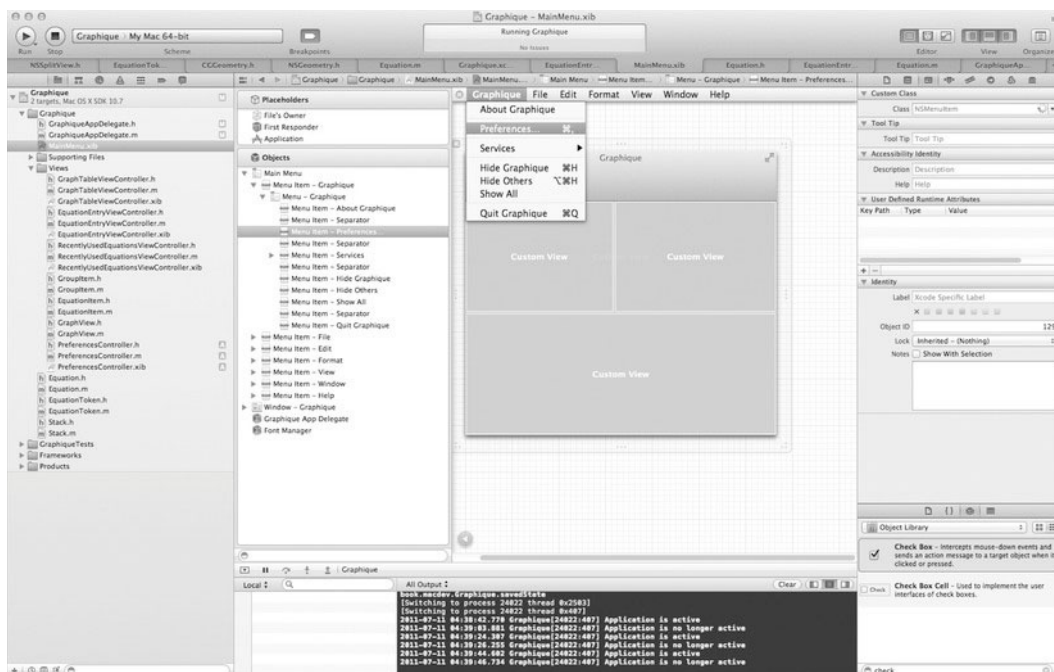


Figure 5-15. Wiring `showPreferencesPanel1:` to the Preferences menu item

Build and run Graphique, and then select Preferences... from the menu. The Preferences window displays, as shown in Figure 5-16.

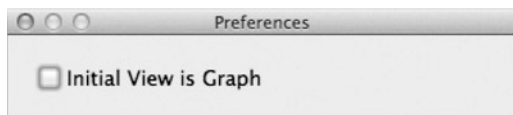


Figure 5-16. The Preferences panel

Check the box, and quit the application to flush the cache. Then, open a Terminal instance and type the following:

```
defaults read book.macdev.Graphique
```

Among the output, you should see a line like this:

```
InitialViewIsGraph = 1;
```

The Preferences panel is working and is writing the preferences to the user defaults. The last step in implementing the custom preference is to actually heed its setting in the application.

Using the Custom Preference

To use the custom preference, we must tell Graphique to check the value for the `InitialViewIsGraph` setting in the user defaults and select the appropriate tab. Note that this happens only on user startup; once a user starts interacting with the application, entering equations and selecting tabs, we leave automatic tab setting alone.

To programmatically select a tab, we need a handle to the tab view in the Graph view. Open `GraphTableViewController.h` and add an `NSTabView` property called `tabView`. Listing 5–10 shows the updated code file.

Listing 5–10. *GraphTableViewController.h*

```
#import <Cocoa/Cocoa.h>

#import "Equation.h"

@class GraphView;

@interface GraphTableViewController : NSViewController <NSTableViewDataSource>

@property (nonatomic, retain) NSMutableArray *values;
@property (weak) IBOutlet NSTableView *graphTableView;
@property (nonatomic, assign) CGFloat interval;
@property (weak) IBOutlet GraphView *graphView;
@property (weak) IBOutlet NSTabView *tabView;

- (void)draw:(Equation *)equation;

@end
```

In `GraphTableViewController.m`, add a `@synthesize` line for `tabView`, and then create an implementation of `awakeFromNib`: that will automatically be called when the user interface loads. In this method, you get a handle to the user defaults, you read the value for `InitialViewIsGraph`, and you select the appropriate tab using `tabView's selectTabViewItemAtIndex:` method. Listing 5–11 shows the code for the `awakeFromNib`: method.

Listing 5–11. *The awakeFromNib: Method That Selects the Proper Tab*

```
- (void)awakeFromNib
{
    // Get the user defaults
    NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

    // Determine which tab to select based on the user defaults
    NSInteger selectedTab = [userDefaults boolForKey:@"InitialViewIsGraph"] ? 0 : 1;

    // Select the proper tab
    [tabView selectTabViewItemAtIndex:selectedTab];
}
```

Finally, connect the Tab View object in Interface Builder to the `tabView` property by selecting `GraphTableViewController.xib` in Xcode, Ctrl+dragging from File's Owner to the Tab view, and selecting `tabView` from the pop-up. Your custom preference should

now function as expected. Launch Graphique, open the Preferences panel, and deselect the check box. Close Graphique and then relaunch it. The Data tab should be selected, as shown in Figure 5–17. You can reselect the check box in the Preferences panel and relaunch Graphique to see the Graph tab selected.

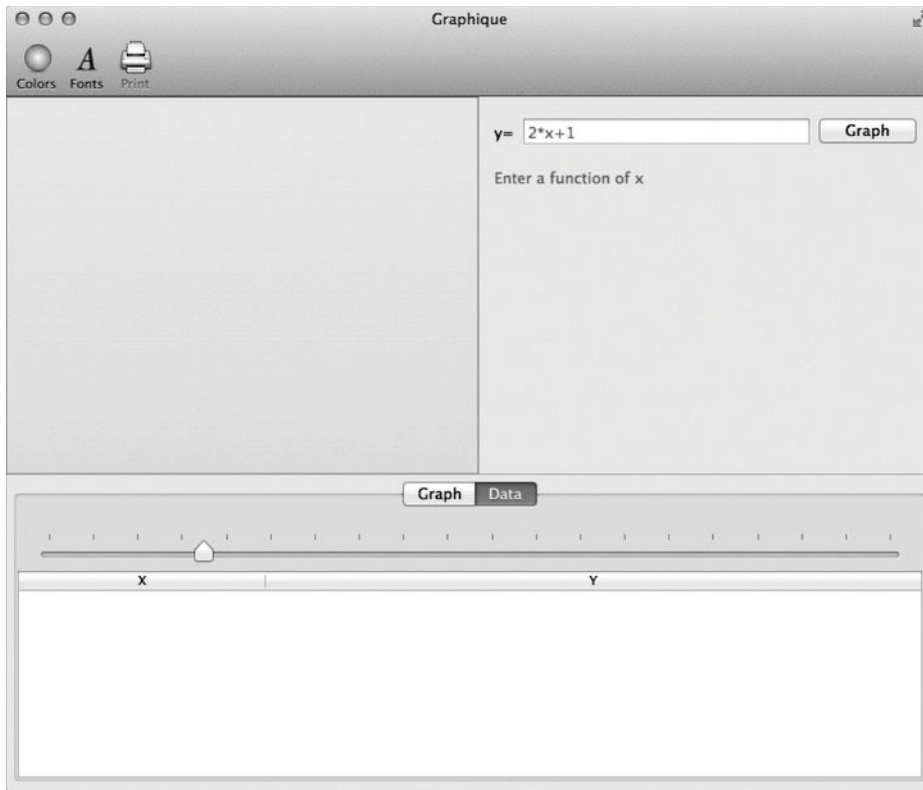


Figure 5–17. *Graphique with the Data tab initially selected*

Using the Local File System

One of the major advantages of running a desktop application is access to local resources. Some common local resources are CPU, memory, and also the file system. The file system used to be synonymous with access to the hard drive. Nowadays, file systems are much more than that. Yes, you can access the local hard drive. But you can also access any mounted remote drives such as Dropbox or WebDAV accounts. All of the access protocols are unified under one file manager API that we review in this section. We then utilize the newly acquired knowledge in Graphique to export graphs as images.

Browsing the File System

In Objective-C, the overt interface to the file system is the `NSFileManager` class. This is the class that gives access to the usual file manipulations. The `NSFileManager` class makes no assumption about what it finds on the file system, so it calls everything (folders, files, symbolic links, and so on) items.

Enumerating Through the Mounted Volumes

To browse the file system, you first need to know what volumes are mounted, and that is done through a simple `NSFileManager` call:

```
NSFileManager *fm = [NSFileManager defaultManager];

NSArray *mountedDisks = [fm mountedVolumeURLsIncludingResourceValuesForKeys:nil
options:NSVolumeEnumerationSkipHiddenVolumes];
NSLog(@"Found %lu volumes", mountedDisks.count);
for(NSURL *path in mountedDisks)
{
    NSLog(@"\t%@", path);
}
```

Running the previous code will provide you with a list of URLs representing the mounted volumes on your system:

```
FileSystem[32607:903] Found 6 volumes
FileSystem[32607:903]   file://localhost/
FileSystem[32607:903]   file://localhost/Volumes/Backup/
FileSystem[32607:903]   file://localhost/Volumes/jdoe/
FileSystem[32607:903]   file://localhost/Volumes/Media/
FileSystem[32607:903]   file://localhost/Volumes/Data/
FileSystem[32607:903]   file://localhost/Volumes/jdoe_HomeDir/
```

Each URL can be used as the starting point for further exploration.

Enumerating Through a Folder

Let's pretend you created a folder called `MyData` directly in your home directory using Finder and copied a few files in there, as shown in Figure 5–18.

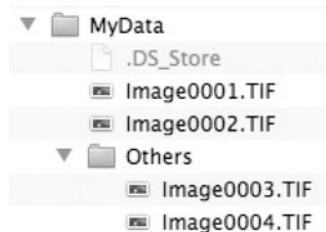


Figure 5–18. A directory with sample files

You would list its contents and whether each item is a file or directory using the following code:

```
NSFileManager *fm = [NSFileManager defaultManager];

NSString* path = [NSHomeDirectory() stringByAppendingPathComponent:@"MyData"];
NSArray *list = [fm contentsOfDirectoryAtURL:[NSURL URLWithString:path]
includingPropertiesForKeys:nil options:NSDirectoryEnumerationSkipsHiddenFiles
error:nil];

NSLog(@"Found %lu items", list.count);
for(NSURL *itemURL in list) {
    BOOL isDirectory;
    [fm fileExistsAtPath:[itemURL path] isDirectory:&isDirectory];

    NSLog(@"\t%@", directory? @"", [itemURL path], (isDirectory ? @"Yes" : @"No"));
}
```

NOTE: We've used the `NSHomeDirectory` function to find the current user's home directory. There are other similar predefined functions such as `NSHomeDirectoryForUser` or `NSTemporaryDirectory`.

If you've created the content as previously illustrated, then the output would be similar to the output shown here:

```
FileSystem[34747:903] Found 3 items
FileSystem[34747:903] /Users/michael/MyData/Image0001.TIF, directory? No
FileSystem[34747:903] /Users/michael/MyData/Image0002.TIF, directory? No
FileSystem[34747:903] /Users/michael/MyData/Others, directory? Yes
```

Notice how only two out of the four image files are listed. This is because the two others are inside the `Others` directory, and if you wanted to list them, you'd have to recurse through the subdirectories.

Writing to the File System

Writing a file to the file system is just as simple. Both `NSString` and `NSData` have methods for writing content to the file system. To write a text file to the file system, the code would look as shown here:

```
NSString *myText = @"This is sample text I would like to store in a file.";
NSString* path = [NSHomeDirectory() stringByAppendingPathComponent:@"MyData"];
NSString *filePath = [path stringByAppendingPathComponent:@"sample.txt"];
[myText writeToFile:filePath atomically:YES
encoding:NSUTF8StringEncodingConversionAllowLossy error:nil];
```

This works fine when the data to write is a string. If the data is binary, however, you must use the `NSData` class. In this case, the code is very similar:

```
NSData *myData = ... // Obtain some data
NSString* path = [NSHomeDirectory() stringByAppendingPathComponent:@"MyData"];
NSString *filePath = [path stringByAppendingPathComponent:@"sample.data"];
[myData writeToFile:filePath atomically:YES];
```

NOTE: Both `writeToFile:` methods have an `atomically` parameter. When that parameter is set to YES, the file content is first written to a temporary location. When the writing is complete, the temporary file is then moved to the new location. This preserves the integrity of the file in case of system interruption during the write operation. If that parameter is set to NO, then the content is written directly to the final location. In the event of an interruption, the file is left in an incomplete state.

Reading from the File System

Reading from the file system is obviously equally as important as writing. This is such a common task that Apple has, once again, added convenience methods to the `NSString` and `NSData` classes.

To read the `sample.txt` file back into a string, use the following code:

```
NSString* path = [NSHomeDirectory() stringByAppendingPathComponent:@"MyData"];
NSString *filePath = [path stringByAppendingPathComponent:@"sample.txt"];
NSString *myText = [NSString stringWithContentsOfFile:filePath
encoding:NSUTF8StringEncoding conversionAllowLossy error:nil];
NSLog(@"Result: %@", myText);
```

Similarly, the data file can be read using the following:

```
NSString* path = [NSHomeDirectory() stringByAppendingPathComponent:@"MyData"];
NSString *filePath = [path stringByAppendingPathComponent:@"sample.data"];
NSData *myData = [NSData dataWithContentsOfFile:filePath];
```

Exporting Graphs as Images

Now that we know how to interact with the file system, we put our newly acquired knowledge to the test. We add a new item in the menu that allows us to export a graph as a PNG image. Upon selecting the menu item, a save dialog will open so that the user can select the export location.

Creating an Image from a View

Before doing anything else, we need to make sure the `GraphTableViewController` is able to produce an image representation of its current graph. Open `GraphTableViewController.h` and declare a new method:

```
-(NSBitmapImageRep*)export;
```

Now open `GraphTableViewController.m` so we can implement the new method we just declared. This is done by painting the image into a bitmap cache and returning that cache.

```
-(NSBitmapImageRep*)export
{
```

```

    NSSize mySize = graphView.bounds.size;

    NSBitmapImageRep *bir = [graphView
    bitmapImageRepForCachingDisplayInRect:graphView.bounds];
    [bir setSize:mySize];
    [graphView cacheDisplayInRect:graphView.bounds toBitmapImageRep:bir];

    return bir;
}

```

Next, open `GraphiqueAppDelegate.h` and declare a new selector to receive the menu item action:

```
-(IBAction)exportAs:(id)sender;
```

Then edit `GraphiqueAppDelegate.m` to implement the new method:

```

-(IBAction)exportAs:(id)sender
{
    // Obtain the image representation
    NSBitmapImageRep* imageRep = [graphTableViewController export];

    // Create the PNG representation
    NSData *data = [imageRep representationUsingType: NSPNGFileType properties: nil];

    // Create the Save As... dialog
    NSSavePanel *saveDlg = [NSSavePanel savePanel];
    [saveDlg setAllowedFileTypes:[NSArray arrayWithObject:@"png"]];

    // Open the dialog and save if the user selected OK
    NSInteger result = [saveDlg runModal];
    if (result == NSOKButton)
    {
        [data writeToURL:saveDlg.URL atomically:YES];
    }
}

```

Calling the `setAllowedFileTypes:` method allows you to specify the file extension for the files you are saving. In this case, we chose `png`. If the users provide a different extension, they are automatically prompted to clarify whether they really want to use a different extension, as Figure 5–19 shows.

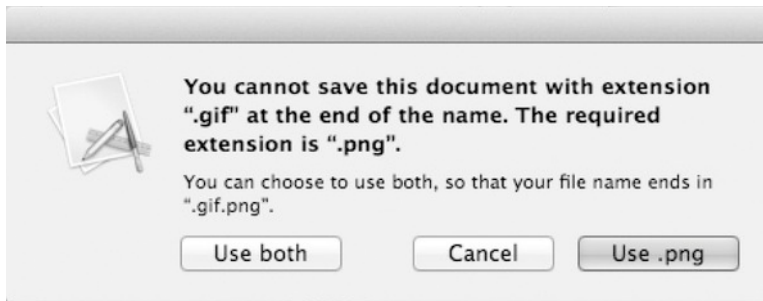


Figure 5–19. An alert generated when the wrong file extension is selected

Adding the Export Graph Menu Item

Our next task is to add the new menu item. Open MainMenu.xib and expand the File menu, as shown in Figure 5–20.

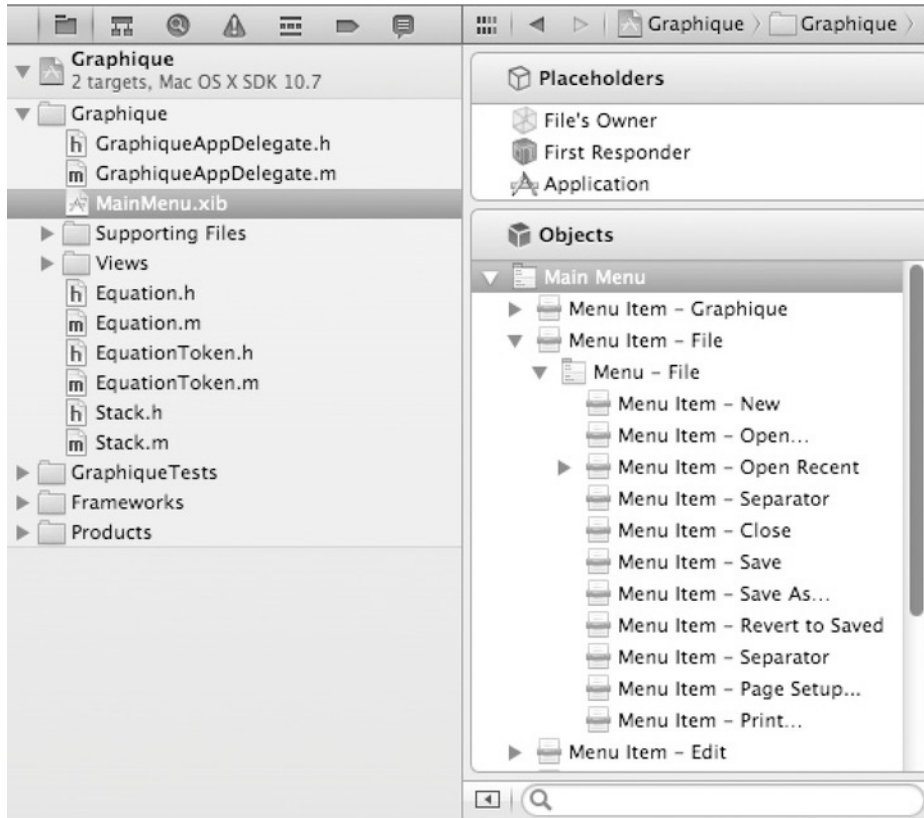


Figure 5–20. *The File menu item expanded*

Find Separator Menu Item in the Object Library and drag it inside the expanded File menu. Then find Menu Item and drag it into the File menu as well, as illustrated in Figure 5–21.

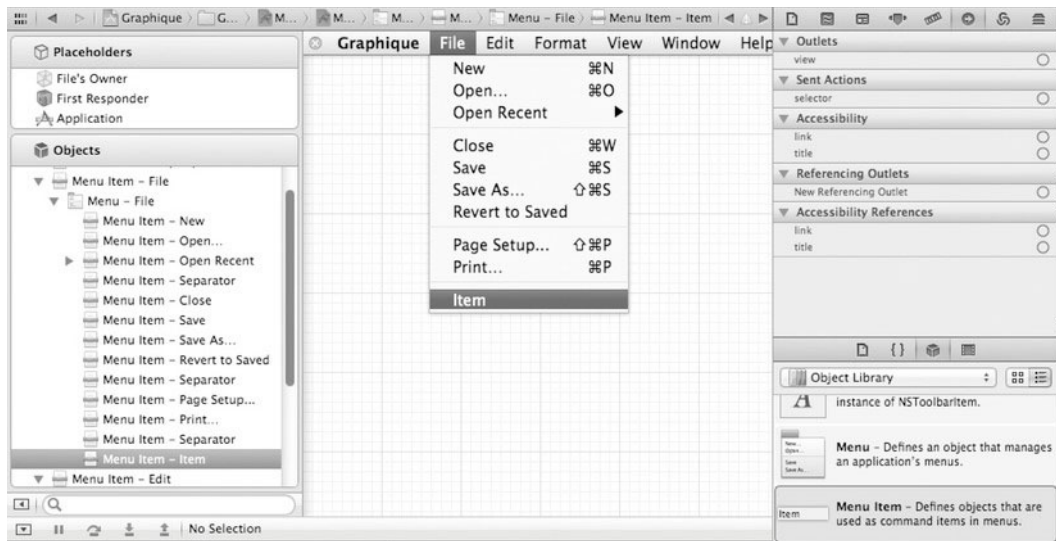


Figure 5-21. *The new menu item*

Open the Attributes inspector and change the menu item title to **Export As...**, as shown in Figure 5-22.

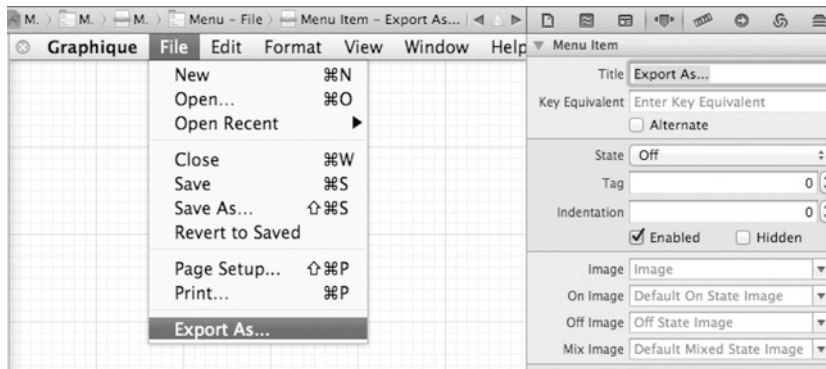


Figure 5-22. *The Export As... menu item*

In the Objects pane, you should already have the Graphique App Delegate object. Ctrl+drag from the **Export As...** menu item to the Graphique App Delegate object to link the action to the `exportAs:` method. The connection should be visible in the Connections Inspector, as shown in Figure 5-23.

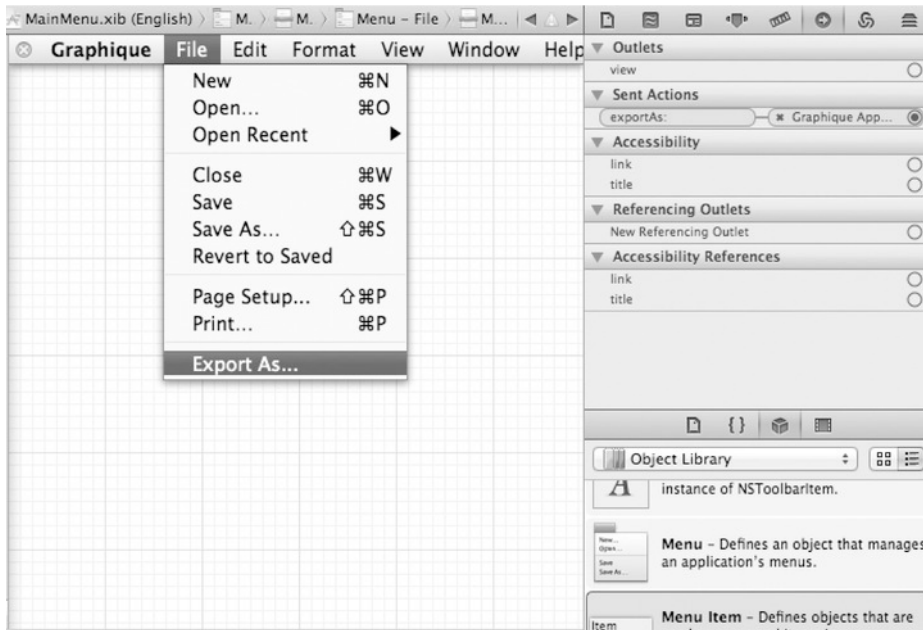


Figure 5–23. *The Export As... item linked*

Run the application to test all you've done. Once the application starts, enter an equation and plot it. Then select **File** ► **Export As...** to see the Save As dialog, as shown in Figure 5–24.



Figure 5–24. *The Save As dialog for exporting graphs*

Summary

In this chapter, you learned how to take the application to the next level by adding some level of persistence. You should have a good understanding of managing user preferences and accessing the file system. Graphique now remembers the user's preference by using `NSUserDefaults`, and we've used these preferences with colors and fonts. It also export graphs as an image to the file system using the `NSFileManager` API.

Using the file system is great for storing unstructured data, such as an image. But it gets a bit more complicated to manage efficiently when the data is structured and has to be editable and even searchable. This is where you need to step up to a persistence framework that can handle structured data. In the next chapter, we expand our persistence capabilities by looking into the Core Data framework.

Using Core Data

Picking up where we left off in Chapter 5, we now get a glimpse of Cocoa's powerful object storage framework, Core Data, which we use to store recently entered equations so that users can retrieve and redisplay them.

At the end of this chapter, Graphique will look like Figure 6–1. It will store the recently entered equations into Core Data storage so that the equations users enter will still be present across launches of Graphique, so that users will be able to retrieve their interesting equations even after they close and reopen Graphique.

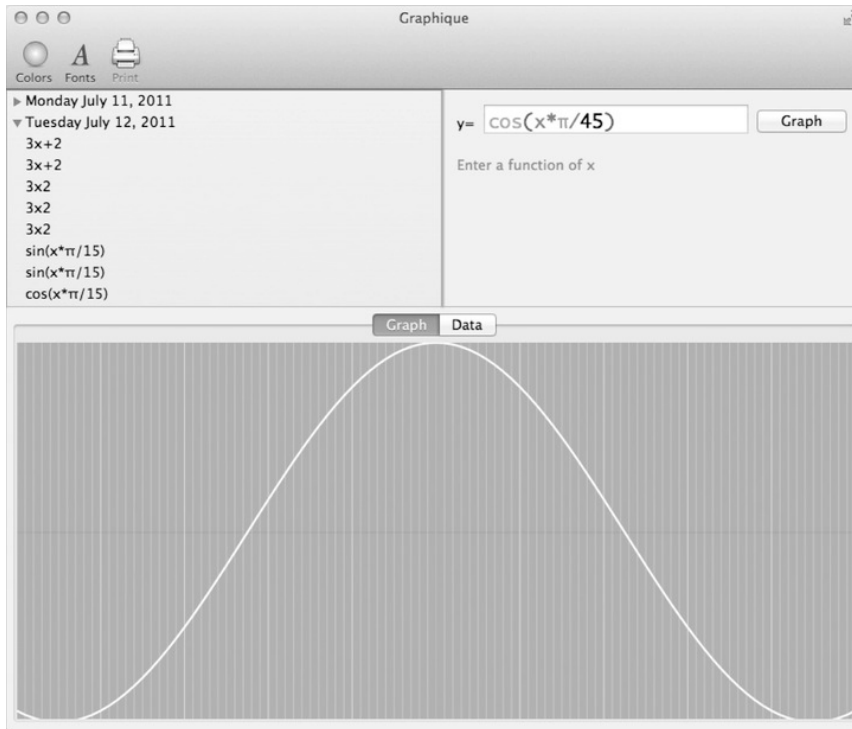


Figure 6–1. *Graphique with equations in Core Data*

Stepping Up to Core Data

Core Data is the persistence framework that comes with Mac OS X. Core Data is not a database, and it's not a file system either. Instead, it's an abstraction layer that stands between your code and the actual storage mechanism (database, file system, and so on) you use. There are many benefits to using Core Data instead of directly welding your code onto your chosen storage mechanism. These are some of them:

- Unified API for storing, retrieving, and searching
- Ability to change storage without changing much of your code
- Automatic bidirectional object to storage mapping
- Ability to use the built-in Xcode tools

In this section, we first go through the steps required to add Core Data to the existing Graphique project. We then create a data model for storing recently used equations. We then hook all this up into the user interface to tie everything together.

Adding Core Data to the Graphique Application

If you've done enough planning and you already know you will use Core Data for your application, you can attach it to your application directly from the New Project wizard when you go through it. In many cases, however, you won't realize until later that you want to use Core Data for your project. In this section, we show you how to add Core Data to an existing project.

Enabling an application to leverage Core Data is a three-step process:

1. Add the Core Data framework.
2. Create a data model.
3. Initialize the managed object context.

We walk you through these three steps so you can add Core Data support to any existing Mac OS X application.

Adding the Core Data Framework

In the Objective-C world, libraries are bundled into frameworks, and an application refers to a framework to gain access to those libraries. To see which frameworks are linked to your application, navigate to the project's build phases with the following steps:

1. Select the Graphique project at the top of the Project navigator.
2. Once the project information is displayed, select the Graphique target.

3. Click the Build Phases tab.
4. Expand the item called Link Binary With Libraries.

Figure 6–2 shows the resulting screen.

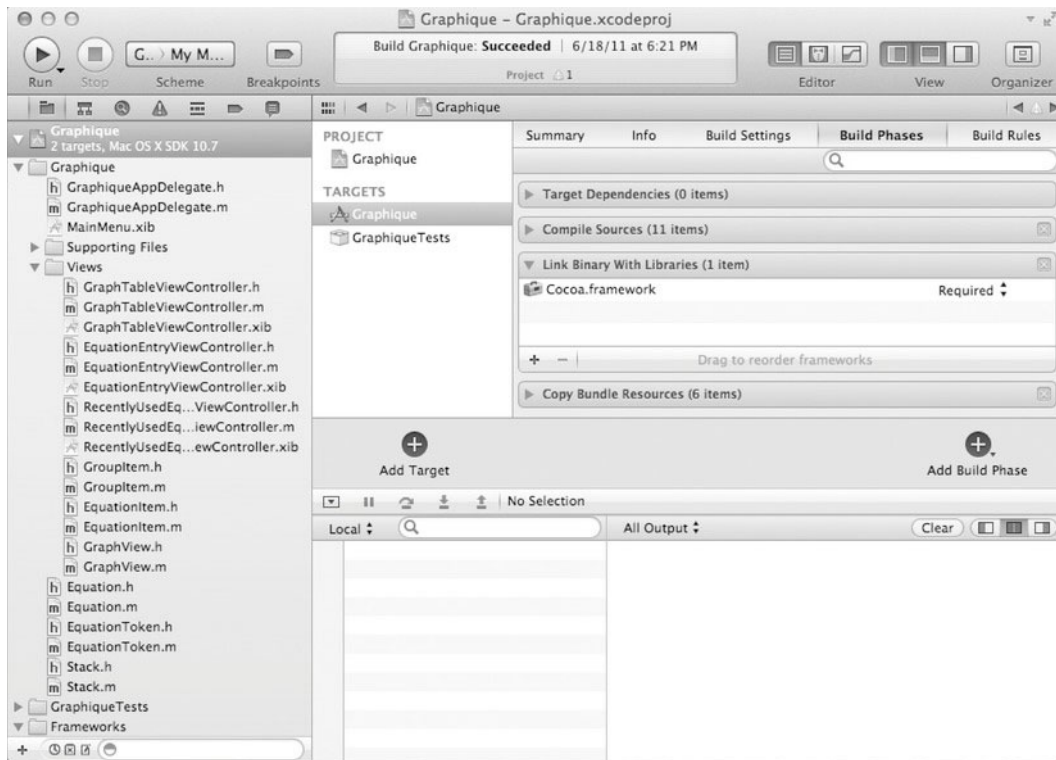


Figure 6–2. *The frameworks linked to the application*

The application should be linked to the Cocoa framework, which is the library that we’ve been using all along and that provides all the windows, buttons, and so on.

Click the + button right below the linked frameworks to add a new framework. When the framework browser opens, look for CoreData.framework and add it to the application, as shown in Figure 6–3.

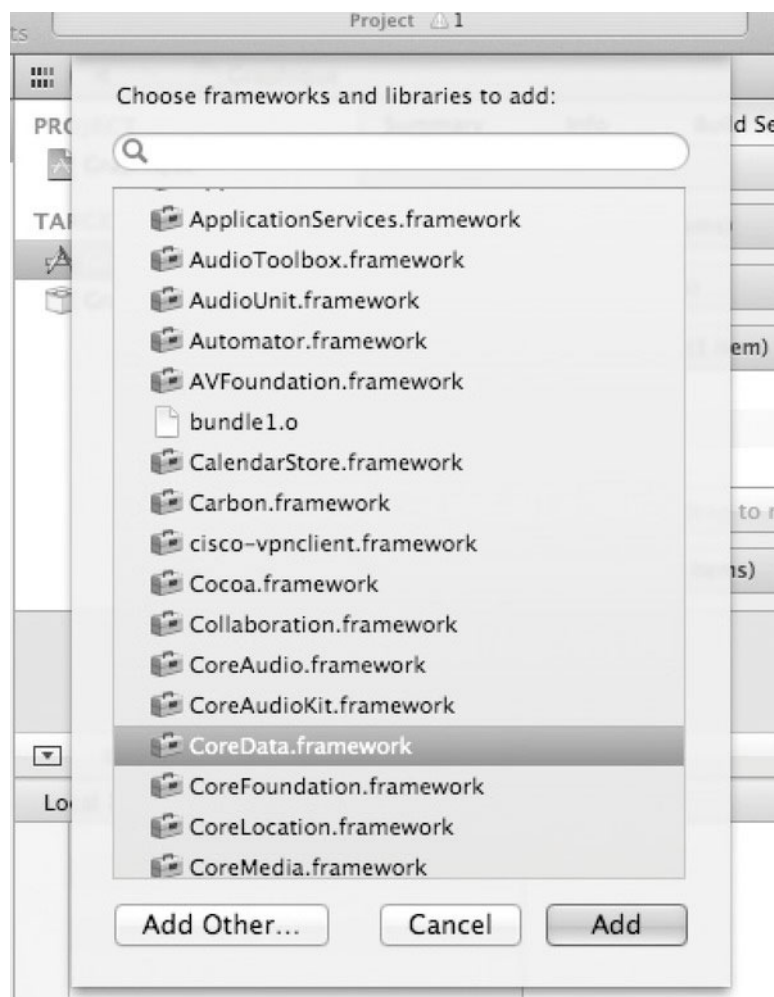


Figure 6–3. Adding the Core Data framework to the application

Once added, CoreData.framework is listed in the linked frameworks, as illustrated in Figure 6–4.

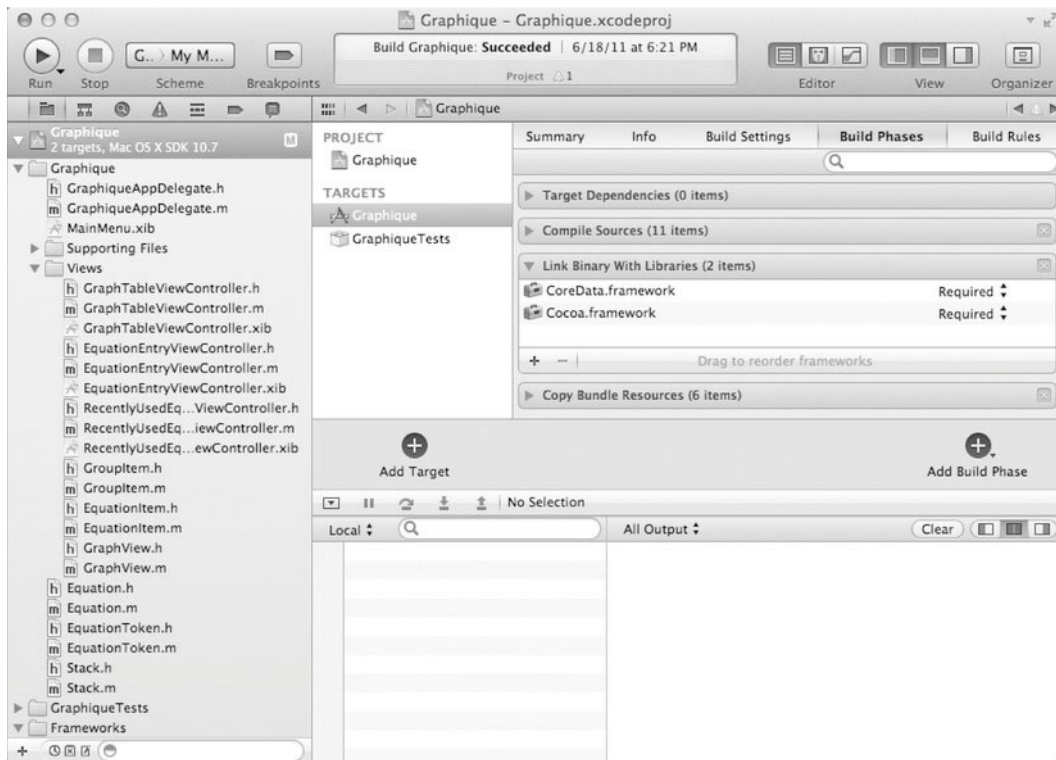


Figure 6–4. The Core Data framework added to the application

After these simple steps, your application is now ready to start tapping into the power of Core Data.

Creating a Data Model

Core Data is a persistence framework. It is a unified bridge between your objects (including the relationships among them) and a data persistence mechanism such as a database or an XML file, for example. When using Core Data, you rarely have to worry about the actual persistence mechanism. Everything rotates around an object model that defines how your data is laid out. Core Data converts that model into a data model if your back end is a database or an XML schema if your back end is an XML file. Regardless of the actual storage mechanism we use, we have to create an object model.

NOTE: Note that we use the term *object model* and not *data model*. *Data model* is a term commonly used in the relational database world to represent the physical layout of the data in a database. This task is left to Core Data. Instead, we define an object model, which serves to define how we want our data objects to relate to one another.

For the Graphique application, we want to use Core Data to store the recently used equations. To keep everything neatly organized, we want to also create groups of equations. Our object model should therefore contain two kinds of objects: Equations and Groups.

Designing the Data Model

Since we added Core Data after the project was already created, we need to create the new object model. Right-click Supporting Files in the Project navigator and select New File, as shown in Figure 6–5.

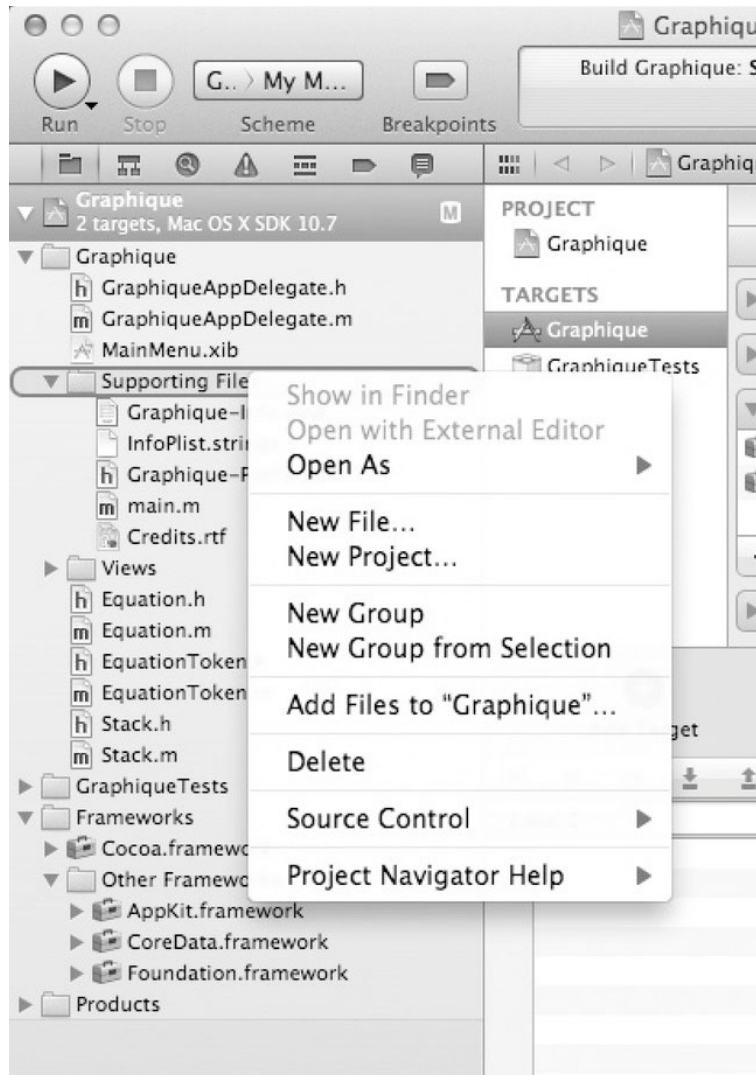


Figure 6–5. Adding a new object model file

From the list of templates, select Data Model from the Mac OS X Core Data category, as illustrated in Figure 6–6.

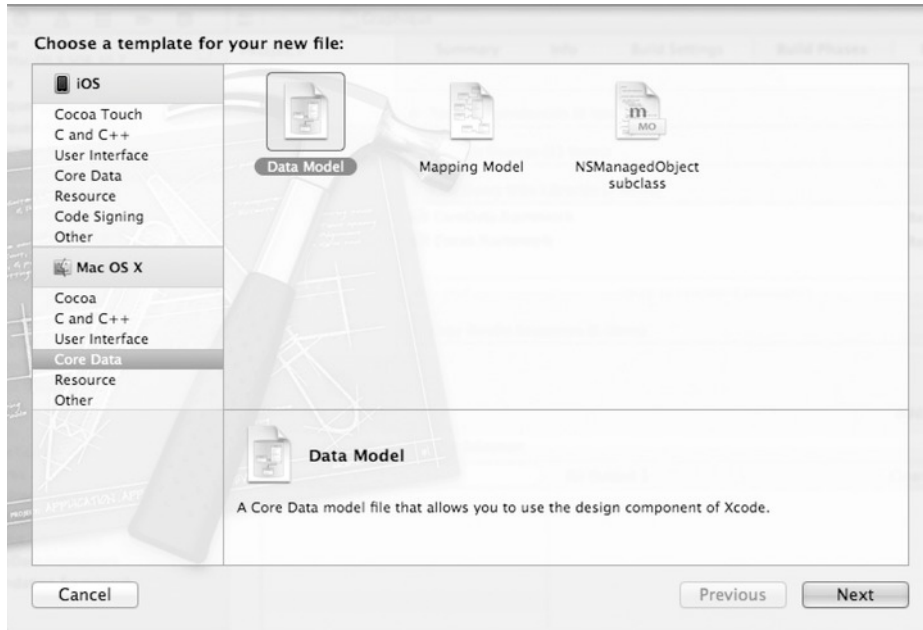


Figure 6–6. Adding a new data model

Save the new file as `GraphiqueModel`, as shown in Figure 6–7, and select the newly created file to see the model editor.

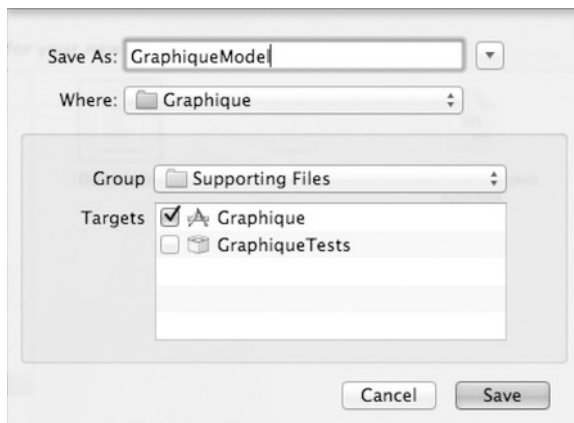


Figure 6–7. Saving the new data model

The most important feature of a data model is the list of entities. Entities represent data objects, in other words, the objects that contain the data and that we want to persist in the data store. Since we just created the model, there is no entity in it. Click

the Add Entity button in the model editor and name the new entity Equation, as illustrated in Figure 6–8.

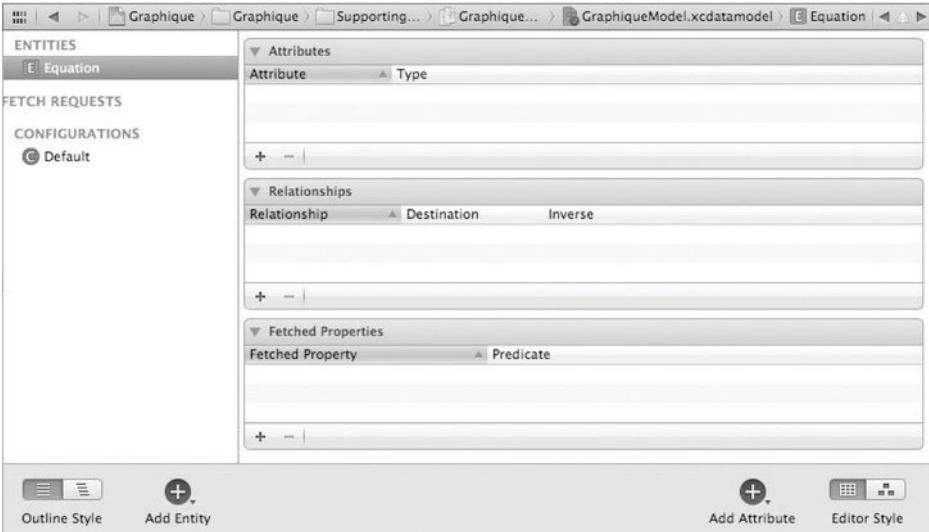


Figure 6–8. Adding a new entity

The list of entities should now contain only an entry for Equation. Much like objects, entities have attributes. Attributes contain the data and can use one of several predefined data types. Table 6–1 lists all the Core Data data types and what Objective-C type they map to.

Table 6–1. The Predefined Core Data Attribute Types

Core Data Attribute Type	Objective-C Data Type	Description
Integer 16	NSNumber	A 16-bit integer
Integer 32	NSNumber	A 32-bit integer
Integer 64	NSNumber	A 64-bit integer
Decimal	NSDecimalNumber	A base-10 subclass of NSNumber
Double	NSNumber	An object wrapper for double
Float	NSNumber	An object wrapper for float
String	NSString	A character string
Boolean	NSNumber	An object wrapper for a boolean value

Core Data Attribute Type	Objective-C Data Type	Description
Date	NSDate	A date object
Binary Data	NSData	Unstructured binary data
Transformable	Any nonstandard type	Any type transformed into a supported type

NOTE: Transformable attributes are a way to tell Core Data that you would like to use a nonsupported data type in your managed object and that you will help Core Data by providing code to transform the attribute data at persist time into a supported type. We don't expand on this more advanced subject in this book.

A recently used equation is defined only by its string representation and a timestamp, so the equation entity will have two attributes: a representation attribute of type String and a timestamp attribute of type Date.

In the Core Data model editor, make sure the Equation entity is selected, and in the Attributes section, click the + button to add a new attribute. Call the attribute representation and set its type to String. Repeat the procedure to add the timestamp attribute, using the Date type this time. The Equation entity should be defined as illustrated in Figure 6–9.

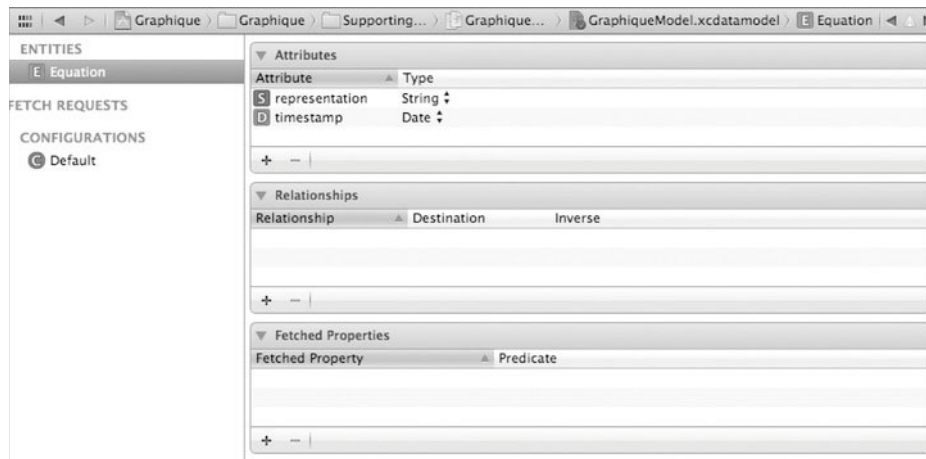


Figure 6–9. The Equation entity

Since we want to use arbitrary groups to create groups of equations, we have to define the Group entity in the same fashion. We define our groups only by a name. In a more evolved application, you may want to include other attributes such as creation time or a description. For now, and to keep things simpler, a group has only a name.

Create the Group entity and add a name attribute of type String, as shown in Figure 6–10.

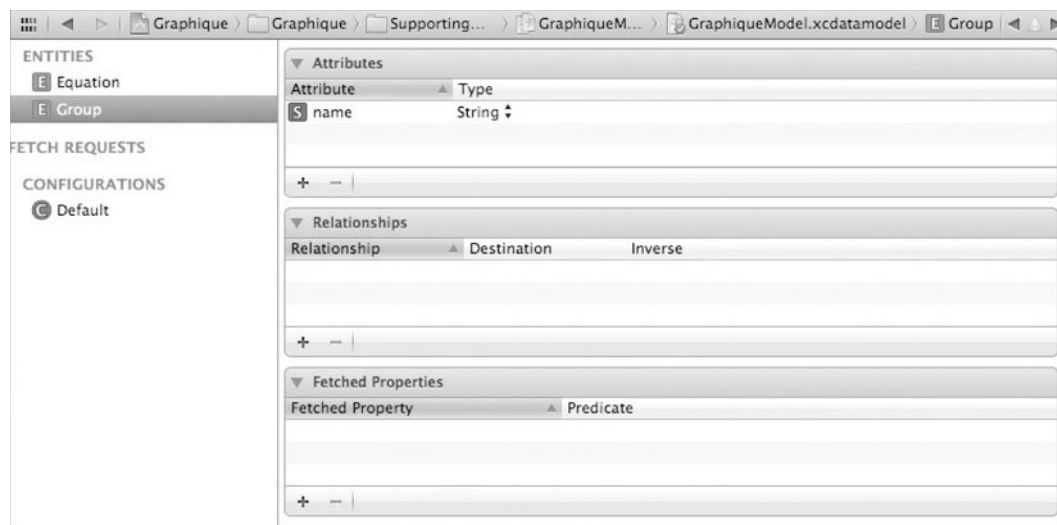


Figure 6–10. *The Group entity*

Near the bottom right of the Core Data model editor, there is an Editor Style button. Use it to toggle from Table view to Graph view. The graph illustrates the relationships between entities. As you can see in Figure 6–11, no relationship currently exists between groups and equations.

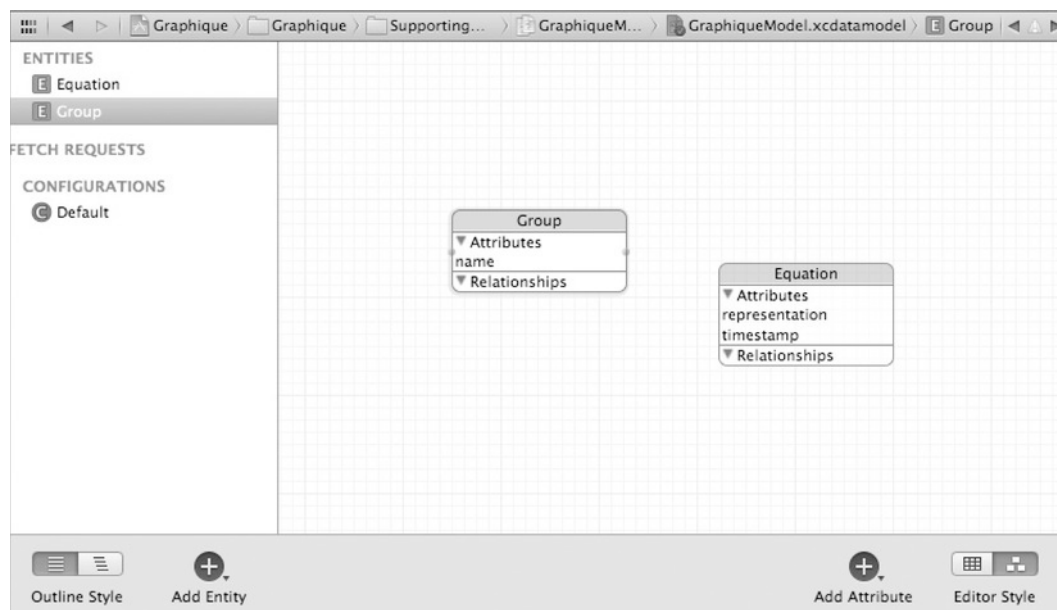


Figure 6–11. *Core Data entities in the graph view*

Since groups contain equations (or equations belong to groups), we need to create the relationship between the two. Switch the editor back to table style. Select the Equation entity and click the + button in the Relationships section. Name the new relationship group and set the destination to Group. This will tell Core Data that an equation might belong to a group. To make sure that Core Data enforces that all equations belong to a group, you must expand the Utilities panel, make sure the newly created relationship is selected, and uncheck the Optional check box, as shown in Figure 6–12.

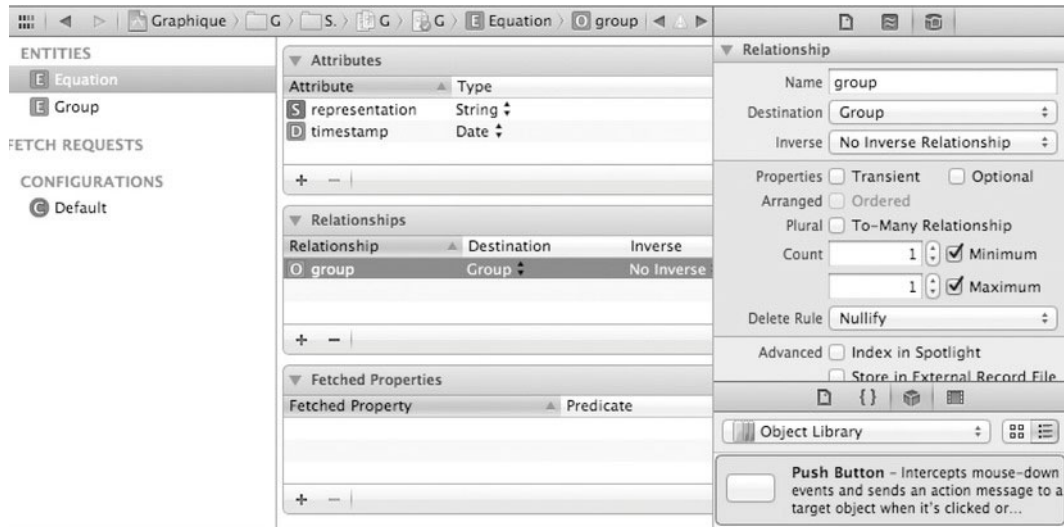


Figure 6–12. Relationship from Equation to Group

To help Core Data manage the object graph, Apple strongly recommends that each relationship have an inverse relationship. For each relationship from entity A to entity B, we must create an inverse relationship from B to A. Select the Group entity and add a new relationship called equations, set its destination to Equation, and its inverse to group. Since a group may contain more than one equation, select To-Many Relationship in the Utilities panel. This tells Core Data that this relationship may lead to multiple equation entities. Since a group may be empty (in other words, does not contain any equations), we leave this relationship as optional. Figure 6–13 shows the resulting relationship configuration.

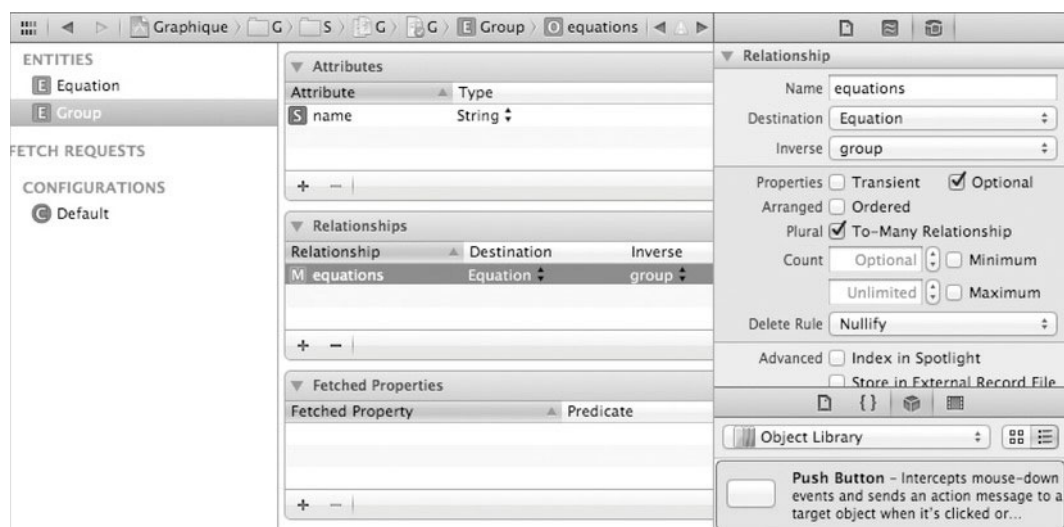


Figure 6–13. The inverse relationship from Group to Equation

This completes our data model for now. If you switch the editor to graph view, as shown in Figure 6–14, you can see that now the two entities are related. Note how the relationship from Group to Equation has a double arrowhead. This is to symbolize the To-many nature of the relationship.

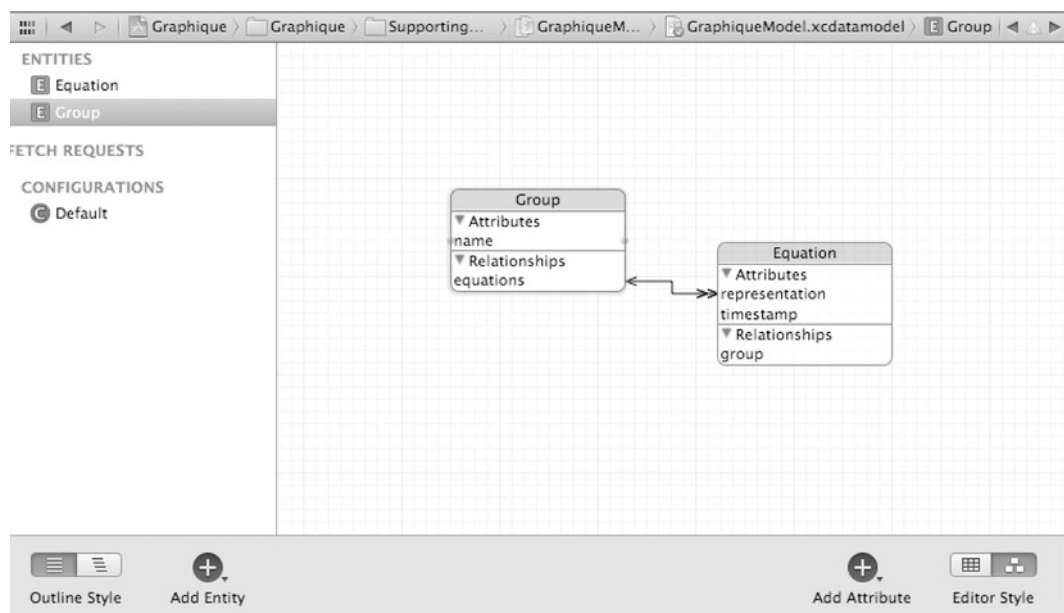


Figure 6–14. The data model in graph view

Initialize the Managed Object Context

Now back to our code. So far, we've made our project aware of Core Data by linking the Core Data framework, and we've defined our data model by creating entities and relationships. When data are pulled out of the persistent store, they are materialized as subclasses of `NSManagedObject` and offer methods to access their data. Any application using Core Data, however, must first initialize the framework before any operation can be performed. The framework uses the `NSManagedObjectContext` class as an interface between your code and the persistent store. The initialization process consists of properly setting up the managed object context and the other classes it relies upon, as shown in Figure 6–15.

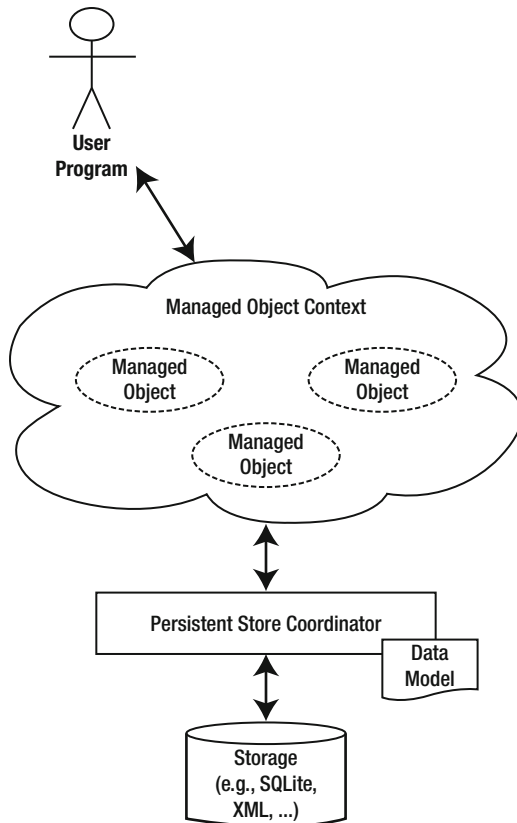


Figure 6–15. *The high-level Core Data framework layout*

The managed object context relies on a persistent store coordinator (`NSPersistentStoreCoordinator`) that serves to reconcile the physical persistent store (database, XML file, and so on) with the object model we created and that gets loaded into the framework as an `NSManagedObjectModel` instance.

The initialization process consists of setting up the `NSManagedObjectModel`, the `NSPersistentStoreCoordinator`, and finally the `NSManagedObjectContext`. For the most

part, these steps are identical from project to project, and we give them to you in the following text. Open `GraphiqueAppDelegate.h` and add an import statement near the top:

```
#import <CoreData/CoreData.h>
```

Define three properties for the three main Core Data components, as illustrated in Listing 6-1: `managedObjectContext`, `managedObjectModel`, and `persistentStoreCoordinator`.

Listing 6-1. *GraphiqueAppDelegate.h with Core Data*

```
#import <Cocoa/Cocoa.h>
#import <CoreData/CoreData.h>

@class EquationEntryViewController;
@class GraphTableViewController;
@class RecentlyUsedEquationsViewController;
@class PreferencesController;

@interface GraphiqueAppDelegate : NSObject <NSApplicationDelegate> {
    @private
    NSManagedObjectContext *managedObjectContext_;
    NSManagedObjectModel *managedObjectModel_;
    NSPersistentStoreCoordinator *persistentStoreCoordinator_;
}

@property (strong) IBOutlet NSWindow *window;
@property (weak) IBOutlet NSSplitView *horizontalSplitView;
@property (weak) IBOutlet NSSplitView *verticalSplitView;
@property (strong) EquationEntryViewController *equationEntryViewController;
@property (strong) GraphTableViewController *graphTableViewController;
@property (strong) RecentlyUsedEquationsViewController
*recentlyUsedEquationsViewController;
@property (strong) PreferencesController *preferencesController;

@property (nonatomic, readonly) NSManagedObjectContext *managedObjectContext;
@property (nonatomic, readonly) NSManagedObjectModel *managedObjectModel;
@property (nonatomic, readonly) NSPersistentStoreCoordinator
*persistentStoreCoordinator;

- (void)changeGraphLineColor:(id)sender;
- (IBAction)showPreferencesPanel:(id)sender;

@end
```

Open `GraphiqueAppDelegate.m` to implement the getter methods for these properties. We start with `managedObjectModel`:

```
- (NSManagedObjectModel *)managedObjectModel {
    if (managedObjectModel_ != nil) {
        return managedObjectModel_;
    }
    managedObjectModel_ = [NSManagedObjectModel mergedModelFromBundles:nil];
    return managedObjectModel_;
}
```

Note that we don't specify the model we created specifically; `mergedModelFromBundles:` will find and load all appropriate model files in the project. Now that we've loaded the

object model, we can leverage it in order to create the persistent store handler. This example uses `NSSQLiteStoreType` in order to indicate that the storage mechanism should rely on a SQLite database, as shown here:

```
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator {
    if (persistentStoreCoordinator_ != nil) {
        return persistentStoreCoordinator_;
    }

    NSString* dir = [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
        NSUserDomainMask, YES) lastObject];
    NSURL *storeURL = [NSURL fileURLWithPath: [dir stringByAppendingPathComponent:
        @"Graphique.sqlite"]];

    NSError *error = nil;
    persistentStoreCoordinator_ = [[NSPersistentStoreCoordinator alloc]
        initWithManagedObjectModel:[self managedObjectModel]];
    if (![persistentStoreCoordinator_ addPersistentStoreWithType:NSSQLiteStoreType
        configuration:nil URL:storeURL options:nil error:&error]) {
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }

    return persistentStoreCoordinator_;
}
```

NOTE: We are telling Core Data to use a SQLite database called `Graphique.sqlite` and store it in the user's Documents folder. You may change this to store the file anywhere you'd like.

Finally, we initialize the context from the persistent store that we just defined:

```
- (NSManagedObjectContext *)managedObjectContext {
    if (managedObjectContext_ != nil) {
        return managedObjectContext_;
    }

    NSPersistentStoreCoordinator *coordinator = [self persistentStoreCoordinator];
    if (coordinator != nil) {
        managedObjectContext_ = [[NSManagedObjectContext alloc] init];
        [managedObjectContext_ setPersistentStoreCoordinator:coordinator];
    }
    return managedObjectContext_;
}
```

The application can now use the managed object context to store and retrieve entities.

Storing Recently Used Equations

In the Graphique application, the `RecentlyUsedEquationsViewController` class is the user interface controller responsible for dealing with recently used equations. Since it will need to interact with the persisted data, the first step is to give it access to the `NSManagedObjectContext` object that represents the interface to the persistent store. Open `RecentlyUsedEquationsViewController.h` and add the following:

1. An `#import <CoreData/CoreData.h>` statement at the top.
2. A new property for holding the managed object context.

Listing 6–2 shows the resulting `RecentlyUsedEquationsViewController.h` file.

Listing 6–2. *RecentlyUsedEquationsViewController.h*

```
#import <Cocoa/Cocoa.h>
#import <CoreData/CoreData.h>

#import "GroupItem.h"

@interface RecentlyUsedEquationsViewController : NSViewController
<NSOutlineViewDataSource, NSSplitViewDelegate> {
@private
    GroupItem *rootItem;
    NSManagedObjectContext *managedObjectContext;
}
@property (nonatomic, strong) NSManagedObjectContext *managedObjectContext;

@end
```

Don't forget to open `RecentlyUsedEquationsViewController.m` and add a `@synthesize` statement for the new property right after the `@implementation` statement:

```
@synthesize managedObjectContext;
```

Finally, we have to make sure the app delegate passes the context along when it initializes the controller. Open `GraphiqueAppDelegate.m` and edit the `applicationDidFinishLaunching:` method to match this code:

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{
    equationEntryViewController = [[EquationEntryViewController alloc]
initWithNibName:@"EquationEntryViewController" bundle:nil];
    [self.verticalSplitView replaceSubview:[self.verticalSplitView subviews]
objectAtIndex:1] with:equationEntryViewController.view];

    graphTableViewController = [[GraphTableViewController alloc]
initWithNibName:@"GraphTableViewController" bundle:nil];
    [self.horizontalSplitView replaceSubview:[self.horizontalSplitView subviews]
objectAtIndex:1] with:[graphTableViewController view];
}
```

```

    recentlyUsedEquationsViewController = [[RecentlyUsedEquationsViewController alloc]
initWithNibName:@"RecentlyUsedEquationsViewController" bundle:nil];
    recentlyUsedEquationsViewController.managedObjectContext = self.managedObjectContext;
    [self.verticalSplitView replaceSubview:[self.verticalSplitView subviews]
objectAtIndex:0] with:[recentlyUsedEquationsViewController view];
    self.verticalSplitView.delegate = recentlyUsedEquationsViewController;

    [[NSColorPanel sharedColorPanel] setTarget:self];
    [[NSColorPanel sharedColorPanel] setAction:@selector(changeGraphLineColor:));
}

```

Now, let's go back to `RecentlyUsedEquationsViewController.h` and define a new method that other controllers can use to tell it to remember an equation:

```
-(void)remember:(Equation*)equation;
```

With the new method, `RecentlyUsedEquationsViewController.h` should look like Listing 6-3.

Listing 6-3. *RecentlyUsedEquationsViewController.h with a New Method for Remembering Equations*

```

#import <Cocoa/Cocoa.h>
#import <CoreData/CoreData.h>

@class GroupItem;
@class Equation;

@interface RecentlyUsedEquationsViewController : NSViewController
<NSOutlineViewDataSource, NSSplitViewDelegate> {
@private
    GroupItem *rootItem;
    NSManagedObjectContext *managedObjectContext;
}
@property (nonatomic, strong) NSManagedObjectContext *managedObjectContext;

- (void)remember:(Equation*)equation;

@end

```

NOTE: Do not confuse the Equation object with the Equation entity we created earlier in the model. Right now, while they represent the same thing semantically, they aren't related in code. Right now, data in the Equation entity is represented as an object of type `NSManagedObject` and not an object of type `Equation`. It is possible to make Core Data use a custom class (in other words, other than `NSManagedObject`) to represent data entities as objects, but it falls out of the scope of this book and encourage you to get a book dedicated to Core Data for more information.

Open `RecentlyUsedEquationsViewController.m` to implement the `remember:` method. The algorithm for the method is as follows:

1. Create a group name based on the current date (month/day/year).
2. Query Core Data to find out whether the group already exists.
3. If it does exist, then use it; otherwise, create a new one, save it, and use it.
4. Create a new equation entity as an `NSManagedObject` and attach it to the appropriate group.
5. Commit to the persistent store.

The next few sections walk you through the implementation for the `remember:` method.

Querying the Persistent Store to Get the Group Entity

Creating the group name is trivial using the `NSDateFormatter` object:

```
NSDate *today = [NSDate date];
NSDateFormatter *dateFormat = [[NSDateFormatter alloc] init];
[dateFormat setDateFormat:@"EEEE MMMM d, YYYY"];
NSString *groupName = [dateFormat stringFromDate:today];
```

We then query the persistent store to see whether the group exists already. Retrieving objects from the persistent store is done using the `NSFetchRequest` object, which defines what to retrieve:

```
// Create the fetch request
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
// Define the kind of entity to look for
NSEntityDescription *entity = [NSEntityDescription entityForName:@"Group"
inManagedObjectContext:self.managedObjectContext];
[fetchRequest setEntity:entity];
// Add a predicate to further specify what we are looking for
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"name=%@", groupName];
[fetchRequest setPredicate:predicate];

NSArray *groups = [self.managedObjectContext executeFetchRequest:fetchRequest
error:nil];
```

```

NSManagedObject *groupMO = nil;

if(groups.count > 0) {
    // We found one, use it
    groupMO = [groups objectAtIndex:0];
}
else {
    // We need to create a new group because it did not exist
    groupMO = [NSEntityDescription insertNewObjectForEntityForName:@"Group"
inManagedObjectContext:self.managedObjectContext];

    // set the name
    [groupMO setValue:groupName forKey:@"name"];
}

```

NOTE: We use the key/value pair generic accessor methods to interact with the data contained in the `NSManagedObject`. `[object valueForKey:@"myAttribute"]` will retrieve the value of `myAttribute`, while `[object setValue:@"theValue" forKey:@"myAttribute"]` will set the value of `myAttribute`.

At this point, we have a managed object representing the right group. If the group did not exist before, we added it using the `insertNewObjectForEntityForName:inManagedObjectContext:` method.

Creating the Equation Managed Object and Adding It to the Persistent Store

Now that we have a valid group managed object, we simply create a new equation managed object and link it to the group:

```

NSManagedObject *equationMO = [NSEntityDescription
insertNewObjectForEntityForName:@"Equation"
inManagedObjectContext:self.managedObjectContext];

// set the timestamp and the representation
[equationMO setValue:equation.text forKey:@"representation"];
[equationMO setValue:[NSDate date] forKey:@"timestamp"];
[equationMO setValue:groupMO forKey:@"group"];

```

Committing

In Core Data, nothing is permanent until you tell it to commit. Upon committing, the managed object context will flush all of its content to the persistent store, making it permanent. The commit operation is triggered by calling the `save:` method:

```

NSError *error = nil;
if (![self.managedObjectContext save:&error]) {
    NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    abort();
}

```

Putting Everything Together into the Final Method

Listing 6–4 documents the entire `remember:` method. Be sure to import `Equation.h` at the top of `RecentlyUsedEquationsViewController.m`.

Listing 6–4. *The `remember:` Method in `RecentlyUsedEquationsViewController.m`*

```
-(void)remember:(Equation*)equation
{
    NSDate *today = [NSDate date];
    NSDateFormatter *dateFormat = [[NSDateFormatter alloc] init];
    [dateFormat setDateFormat:@"EEEE MMMM d, YYYY"];
    NSString *groupName = [dateFormat stringFromDate:today];

    // Create the fetch request
    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    // Define the kind of entity to look for
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Group"
inManagedObjectContext:self.managedObjectContext];
    [fetchRequest setEntity:entity];
    // Add a predicate to further specify what we are looking for
    NSPredicate *predicate = [NSPredicate predicateWithFormat:@"name=%@", groupName];
    [fetchRequest setPredicate:predicate];

    NSArray *groups = [self.managedObjectContext executeFetchRequest:fetchRequest
error:nil];
    NSManagedObject *groupMO = nil;

    if(groups.count > 0) {
        // We found one, use it
        groupMO = [groups objectAtIndex:0];
    }
    else {
        // We need to create a new group because it did not exist
        groupMO = [NSEntityDescription insertNewObjectForEntityForName:@"Group"
inManagedObjectContext:self.managedObjectContext];

        // set the name
        [groupMO setValue:groupName forKey:@"name"];
    }

    NSManagedObject *equationMO = [NSEntityDescription
insertNewObjectForEntityForName:@"Equation"
inManagedObjectContext:self.managedObjectContext];

    // set the timestamp and the representation
    [equationMO setValue:equation.text forKey:@"representation"];
    [equationMO setValue:[NSDate date] forKey:@"timestamp"];
    [equationMO setValue:groupMO forKey:@"group"];

    NSError *error = nil;
    if (![self.managedObjectContext save:&error]) {
        NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }
}
```

Finally, we must call this method. We want to remember an equation when it is entered. Open `EquationEntryViewController.m` and add an import statement at the top:

```
#import "RecentlyUsedEquationsViewController.h"
```

Then, edit the `equationEntered:` method to match Listing 6–5.

Listing 6–5. The updated `equationEntered:` Method

```
- (IBAction)equationEntered:(id)sender
{
    NSLog(@"Equation entered");
    GraphiqueAppDelegate *delegate = [UIApplication sharedApplication].delegate;

    Equation *equation = [[Equation alloc] initWithString: [self.textField stringValue]];

    NSError *error = nil;
    if (![equation validate:&error])
    {
        // Validation failed, display the error
        UIAlertView *alert = [[[UIAlertView alloc] init] autorelease];
        [alert addButtonWithTitle:@"OK"];
        [alert setMessageText:@"Something went wrong. "];
        [alert setInformativeText:[NSString stringWithFormat:@"Error %d: %@", [error
code], [error localizedDescription]]];
        [alert setAlertStyle:NSInformationalAlertStyle];

        [alert beginSheetModalForWindow:delegate.window modalDelegate:self
didEndSelector:@selector(alertDidEnd:returnCode:contextInfo:) contextInfo:nil];
    }
    else
    {
        [delegate.recentlyUsedEquationsViewController remember: equation];
        [delegate.graphTableViewController draw: equation];
    }
}
```

Reloading Recently Used Equations

We have everything in place to store equations as they are entered. Of course, we haven't wired the outline view with the data, so the application won't appear to be doing anything different. You can launch the application nonetheless to check that everything is hooked up correctly.

Once the application is started, try typing an equation like x^2 (in other words, “ x squared”) for example and graph it. Everything should behave like before. But open the Terminal.app application and go into the Documents folder to open the data store:

```
sqlite3 ~/Documents/Graphique.sqlite
SQLite version 3.7.5
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .schema
CREATE TABLE ZEQUATION ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER, ZGROUP
INTEGER, ZTIMESTAMP TIMESTAMP, ZREPRESENTATION VARCHAR );
```

```

CREATE TABLE ZGROUP ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER, Z_OPT INTEGER, ZNAME
VARCHAR );
CREATE TABLE Z_METADATA (Z_VERSION INTEGER PRIMARY KEY, Z_UUID VARCHAR(255), Z_PLIST
BLOB);
CREATE TABLE Z_PRIMARYKEY (Z_ENT INTEGER PRIMARY KEY, Z_NAME VARCHAR, Z_SUPER INTEGER,
Z_MAX INTEGER);
CREATE INDEX ZEQUATION_ZGROUP_INDEX ON ZEQUATION (ZGROUP);
sqlite> select * from ZGROUP;
1|2|1|Wednesday July 6, 2011
sqlite> select * from ZEQUATION;
1|1|1|331670216.49823|x2

```

Use the `.schema` command to display the schema. You can see that Core Data created two tables to hold the equations and group. By querying each table, you can see that a new group was created and that it contains the equation you just typed. The data is getting stored, and now we just have to pull it back out and display it properly.

Before we hook everything to the user interface, we need to edit the `GroupItem` and `EquationItem` objects. These objects are used as the data structure of the outline view we want to populate. First, we edit `EquationItem` to make `text` a property instead of a method that returns dummy data. Edit `EquationItem.h` and `EquationItem.m` to look like Listing 6–6 and Listing 6–7, respectively.

Listing 6–6. *EquationItem.h*

```

#import <Foundation/Foundation.h>

@interface EquationItem : NSObject {
    @private
    NSString *text;
}
@property (nonatomic, strong) NSString *text;

@end

```

Listing 6–7. *EquationItem.m*

```

#import "EquationItem.h"

@implementation EquationItem

@synthesize text;

- (NSInteger)numberOfChildren {
    return 0;
}

@end

```

Next we want to make sure we can reload the data in a group item without having to throw away the object each time, so we add a `reset:` method that will remove its children and a boolean flag to help us keep track of whether the members of the group have been loaded. Edit `GroupItem.h` and `GroupItem.m` to match Listing 6–8 and Listing 6–9, respectively.

Listing 6–8. *GroupItem.h*

```
#import <Foundation/Foundation.h>

@interface GroupItem : NSObject {
@private
    NSString *name;
    NSMutableArray *children;
    BOOL loaded;
}

@property (nonatomic, strong) NSString *name;
@property BOOL loaded;

- (NSInteger)numberOfChildren;
- (id)childAtIndex:(NSUInteger)n;
- (NSString*)text;

- (void)addChild:(id)childNode;
- (void)reset;

@end
```

Listing 6–9. *GroupItem.m*

```
#import "GroupItem.h"

@implementation GroupItem

@synthesize name, loaded;

- (id)init
{
    self = [super init];
    if (self) {
        children = [[NSMutableArray alloc] init];
        loaded = NO;
    }
    return self;
}

- (void)addChild:(id)childNode
{
    [children addObject:childNode];
}

- (NSInteger)numberOfChildren
{
    return children.count;
}

- (id)childAtIndex:(NSUInteger)n
{
    return [children objectAtIndex:n];
}

- (void)reset
{

```

```

    [children removeAllObjects];
    loaded = NO;
}

- (NSString*)text
{
    return name;
}

- (void)dealloc
{
    [children release];
    [super dealloc];
}

@end

```

We now turn our interest toward the `RecentlyUsedEquationsViewController` class. To help read the data from the persistent store, we add a method that will take an item, either a `GroupItem` or an `EquationItem`, and populate its children if it's a group. If the item is an equation, nothing is done since equations are leaves in our outline tree. Open `RecentlyUsedEquationsViewController.h` and declare the following method:

```
-(void)loadChildrenForItem:(id)item;
```

Then edit `RecentlyUsedEquationsViewController.m` to provide the method implementation, as shown in Listing 6–10.

Listing 6–10. The `loadChildrenForItem:` Method

```

-(void)loadChildrenForItem:(id)item
{
    // If the item isn't a group, there's nothing to load
    if (![item isKindOfClass:GroupItem.class]) return;
    GroupItem *group = (GroupItem*)item;

    // No point reloading if it's already been loaded
    if (group.loaded) return;

    // Wipe out the nodes children since we're about to reload them
    [group reset];

    // If the group is the rootItem, then we need to load all the available groups. If
    not, then we only load the
    // equations for that group based on its name

    if (group == rootItem) {
        NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
        NSEntityDescription *entity = [NSEntityDescription entityForName:@"Group"
inManagedObjectContext:self.managedObjectContext];
        [fetchRequest setEntity:entity];

        NSArray *groups = [self.managedObjectContext executeFetchRequest:fetchRequest
error:nil];
        for (NSManagedObject *obj in groups) {
            GroupItem *groupItem = [[GroupItem alloc] init];
            groupItem.name = [obj valueForKey:@"name"];

```

```

        [group addChild:groupItem];
    }

    }
    else {
        NSFetchedRequest *fetchRequest = [[NSFetchedRequest alloc] init];
        NSEntityDescription *entity = [NSEntityDescription entityForName:@"Equation"
inManagedObjectContext:self.managedObjectContext];
        [fetchRequest setEntity:entity];
        // Add a predicate to further specify what we are looking for
        NSPredicate *predicate = [NSPredicate predicateWithFormat:@"group.name=%@",
group.name];
        [fetchRequest setPredicate:predicate];

        NSArray *equations = [self.managedObjectContext executeFetchRequest:fetchRequest
error:nil];

        for(NSManagedObject *obj in equations) {
            EquationItem *equationItem = [[EquationItem alloc] init];
            equationItem.text = [obj valueForKey:@"representation"];
            [group addChild:equationItem];
        }
    }

    // Mark the group as properly loaded
    group.loaded = YES;
}

```

The previous method loads only the data for a given node. It does not recurse down the children to load their data as well. This is commonly referred to as *lazy loading*. Lazy loading is a technique that consists in loading only the data necessary to display the user interface. We make sure to call the previous method only when the user interface needs the information. The information is needed when the outline view wants to determine whether a node can be expanded (in other words, a group with equations) or to find out how many children a node has (in other words, how many equations in a given group). While still in `RecentlyUsedEquationsViewController.m`, edit the `outlineView:numberOfChildrenOfItem:` and `outlineView:isItemExpandable:` methods to match Listing 6–11 and Listing 6–12.

Listing 6–11. Loading a Group When the Number of Children Is Needed

```

- (NSInteger)outlineView:(NSOutlineView *)outlineView numberOfChildrenOfItem:(id)item
{
    [self loadChildrenForItem:(item == nil ? rootItem : item)];
    return (item == nil) ? [rootItem numberOfChildren] : [item numberOfChildren];
}

```

Listing 6–12. Loading a Group to Find Out If the Group Is Expandable

```

- (BOOL)outlineView:(NSOutlineView *)_outlineView isItemExpandable:(id)item
{
    return [self outlineView:_outlineView numberOfChildrenOfItem:item] > 0;
}

```

Of course, we need to make sure we remove any dummy data we had created in a prior chapter. Clean up `initWithNibName:bundle:` to make it look like Listing 6–13.

Listing 6–13. *The initWithNibName:bundle: Method*

```
- (id)initWithNibName:(NSString *)nibNameOrNil bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];
    if (self) {
        rootItem = [[GroupItem alloc] init];
    }
    return self;
}
```

If you launch the application now, the outline view should load the equation you had recorded in the previous run, but if you graph another equation, despite that it is recorded, the table does not refresh itself. To address this issue, we need to make the controller aware of the outline view so it can tell it to refresh. To do this, we add a new property of type `NSOutlineView` and tag it with `IBOutlet` so Interface Builder can see it. `RecentlyUsedEquationsViewController.h` should match Listing 6–14.

Listing 6–14. *RecentlyUsedEquationsViewController.h*

```
#import <Cocoa/Cocoa.h>
#import <CoreData/CoreData.h>

#import "Equation.h"
#import "GroupItem.h"

@interface RecentlyUsedEquationsViewController : NSViewController
<NSOutlineViewDataSource, NSSplitViewDelegate>
{
    @private
        GroupItem *rootItem;
        NSManagedObjectContext *managedObjectContext;
        NSOutlineView *outlineView;
}
@property (nonatomic, strong) NSManagedObjectContext *managedObjectContext;
@property (nonatomic, strong) IBOutlet NSOutlineView *outlineView;

-(void)remember:(Equation*)equation;
-(void)loadChildrenForItem:(id)item;

@end
```

Make sure you open `RecentlyUsedEquationsViewController.m` to add the `@synthesize` directive after `@implementation`:

```
@synthesize outlineView;
```

Finally, open `RecentlyUsedEquationsViewController.xib` and link the new `outlineView` property from the File's Owner to the Outline View object. Select the File's Owner, open the Connections inspector, and drag the circle next to the `outlineView` outlet to the Outline View object. The result should look like Figure 6–16.

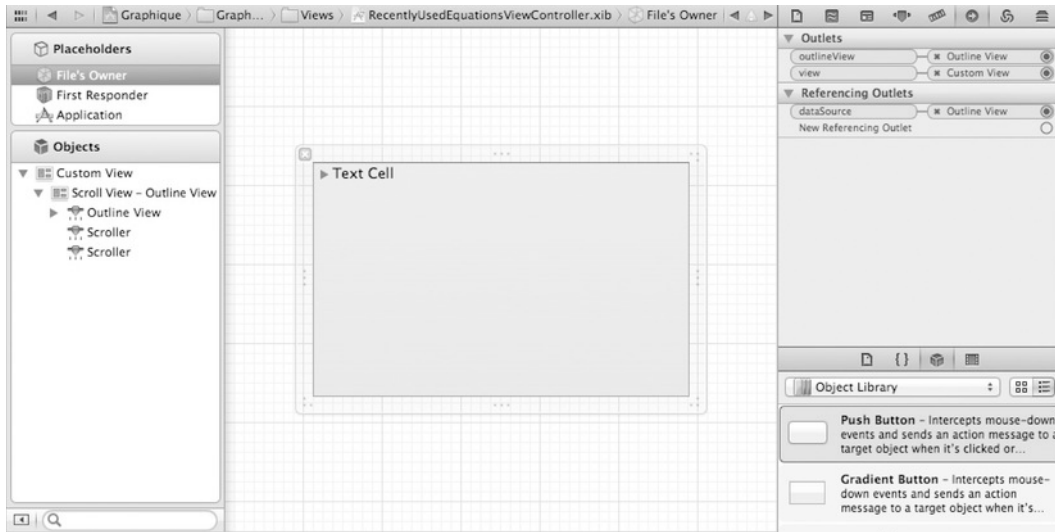


Figure 6–16. *The `outlineView` outlet linked to the user interface*

All we have to do now is tell the outline view to reload itself when we are told to remember an equation. In `RecentlyUsedEquationsViewController.m`, add the following couple of lines of code at the end of the `remember:` method:

```
// Reload outline
[rootItem reset];
[outlineView reloadData];
```

Launch the app and see how it now remembers equations as you graph them.

Tightening the Control over the Outline View

So far, we have loaded our data into the outline view. Now we need to deal with the user interacting with the view. Much like most other Cocoa components, outline views use a delegate to handle interactions. We now set our controller as the delegate for the outline view and implement the necessary methods to catch the events we care about.

Using `NSOutlineViewDelegate`

If you've followed this book page by page and notice the naming patterns, you most likely already guessed that the outline view delegates must conform to the `NSOutlineViewDelegate` protocol. This protocol has no required methods. Edit the interface definition of `RecentlyUsedEquationsViewController.h` to add `NSOutlineViewDelegate` in the list of protocols it conforms to:

```
@interface RecentlyUsedEquationsViewController : NSViewController
<NSOutlineViewDataSource, NSSplitViewDelegate, NSOutlineViewDelegate>
```

All that is left to do is to edit `RecentlyUsedEquationsViewController.xib`, select the Outline View object, and connect its delegate to the controller (that is, the File's Owner). The resulting connection should look like Figure 6–17.

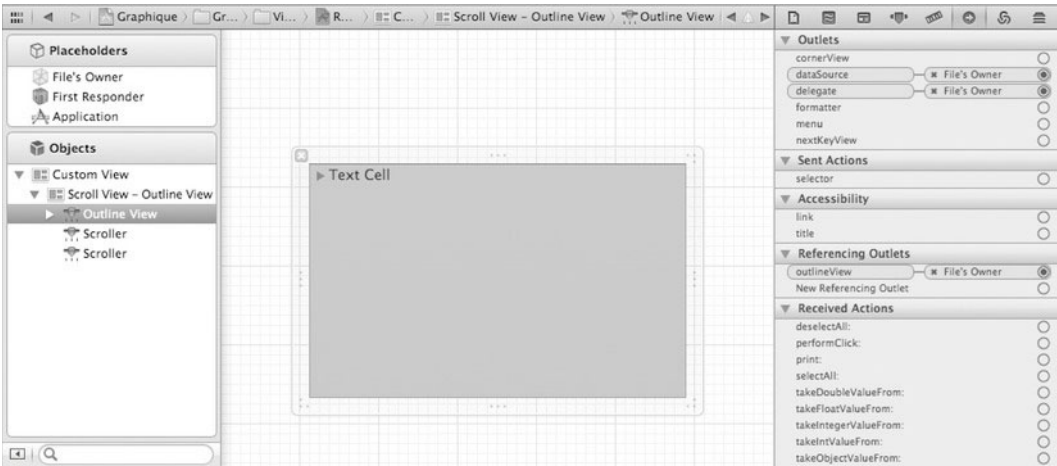


Figure 6–17. The `NSOutlineView` delegate properly connected

At this point, the outline view is told to send all events to its delegate, the `RecentlyUsedEquationsViewController` controller. We can now intercept the events we care about and do what we need to do.

Handling Equations Selection

The most obvious use of the recently used equations is to retrace them by selecting them from the outline view. In addition to the methods it declares, the delegate protocol is automatically registered to receive messages corresponding to `NSOutlineView` notifications. These inform the delegate when the selection changes or is about to change, when a column is moved or resized, and when an item is expanded or collapsed. See Table 5-2 for the delegate messages and the corresponding notification types.

Table 5-2. `NSOutlineView` Delegate Messages and Notifications

Delegate Message	Notification
<code>outlineViewColumnDidMove:</code>	<code>NSOutlineViewColumnDidMoveNotification</code>
<code>outlineViewColumnDidResize:</code>	<code>NSOutlineViewColumnDidResizeNotification</code>
<code>outlineViewSelectionDidChange:</code>	<code>NSOutlineViewSelectionDidChangeNotification</code>
<code>outlineViewSelectionIsChanging:</code>	<code>NSOutlineViewSelectionIsChangingNotification</code>
<code>outlineViewItemDidExpand:</code>	<code>NSOutlineViewItemDidExpandNotification</code>
<code>outlineViewItemDidCollapse:</code>	<code>NSOutlineViewItemDidCollapseNotification</code>

To catch the selection change, we provide an implementation of the `outlineViewSelectionDidChange:` method in `RecentlyUsedEquationsViewController.m`, as shown in Listing 6–15.

Listing 6–15. Handling Selection Change in an Outline View

```
- (void)outlineViewSelectionDidChange:(NSNotification *)notification
{
    NSOutlineView *outlineView_ = [notification object];
    NSInteger row = [outlineView_ selectedRow];

    id item = [outlineView_ itemAtRow:row];

    // If an equation was selected, deal with it
    if([item isKindOfClass:EquationItem.class]) {
        EquationItem *equationItem = item;

        Equation *equation = [[Equation alloc] initWithString:equationItem.text];

        GraphiqueAppDelegate *delegate = [UIApplication sharedApplication].delegate;

        [delegate.equationEntryViewController.textField setStringValue: equation.text];
        [delegate.graphTableViewViewController draw:equation];

        [delegate.equationEntryViewController controlTextDidChange: nil];
    }
}
```

Make sure you add the proper import statements at the top of `RecentlyUsedEquationsViewController.m`:

```
#import "GraphiqueAppDelegate.h"
#import "EquationEntryViewController.h"
#import "GraphTableViewViewController.h"
```

Preventing Double-Clicks from Editing

Our outline view is populated from the database, so we don't want users to be able to edit its nodes by double-clicking them. An outline view always asks its delegate before allowing a cell to be edited. All we have to do is make sure we always say no. Add an implementation for the appropriate method to `RecentlyUsedEquationsViewController.m`, as shown here:

```
- (BOOL)outlineView:(NSOutlineView *)outlineView shouldEditTableColumn:(NSTableColumn *)tableColumn item:(id)item
{
    return NO;
}
```

Launch the app and try selecting an equation from the recently used equations. It populates the entry field and draws the equation as expected.

Summary

In this chapter, you learned about Cocoa's powerful object storage mechanism, Core Data, and got a taste for how to use it. Feel free to experiment with Core Data and read further on the topic to add power to your applications. Don't be fooled by the simplicity of the API; Core Data has much more to offer and should be on your mind any time you think about storing structured application data. We strongly encourage you to pick up a book dedicated to the subject.

In the next chapter, we integrate Graphique more fully with the Mac OS X Lion desktop so that it takes advantage of what the Lion environment offers.

Integrating Graphique into the Mac OS X Desktop

Right now, Graphique is an application island in the sea of the Mac OS X desktop. You can launch it, run it, use it fruitfully, and close it, and it's all self-contained. There's nothing wrong with that, but Mac users have grown to expect more from an application: they want it to work with the rest of the Mac OS X ecosystem. In this chapter, you integrate Graphique into that ecosystem, expanding its borders beyond the current shoreline into Finder and the menu bar. At the end of this chapter, you'll be able to save your equations to files, and then double-click them in Finder to launch Graphique and display their graphs. You'll also be able to use Quick Look to see your graphs from within Finder. Finally, you'll be able to display an icon in the menu bar and launch the most recent ten equations from it.

Dealing with Graphique XML Files

Most applications produce data that they store in files. For example, Microsoft Word produces .doc files that we commonly refer to as *Word files*. The expectation is that when you click to open a Word file, the operating system will know to open Microsoft Word. The same thing happens to Photoshop files, Pages files, and countless other types of files. This section illustrates how to create "Graphique files." It then registers the newly created file type with the OS and the file preview capability built in Mac OS Finder.

Producing a Graphique File

At any time, Graphique should be able to save an equation to a Graphique file, which is a file with a .graphique extension. Rather than defining a new file format for Graphique files, though, you'll leverage Mac OS X's support for property list files and store Graphique files as property lists. The first step toward saving a Graphique file is to create a method to respond to user requests to save an equation.

From a design perspective, typically an application has both Save and Save As menu items. Before a document has ever been saved, the application has no idea where to save it and enables only the Save As menu item. When users select this menu item, they must specify a location to save the file. At this point, the application enables the Save menu item, which, when selected, saves the open document to the location already specified.

With Graphique, an equation is a short snippet of text. If a user changes the text, are they modifying the original equation as another version of itself, in which case subsequent Save operations should overwrite the original equation? Or are they typing in a new equation, in which case the application should prompt for a new location to save the file each time? We've opted for the latter, since equations are so short that we think people are more likely to type in new equations rather than fine-tune existing equations, so we implement only the Save As method.

Creating a Save As Method

Open MainMenu.xib and expand the File menu. You can see that Xcode generated only a Save menu item and not a Save As menu item. You're going to change the Save to Save As. Select the Save menu item, open the Attributes inspector, and change the title to Save As.... Select the Key Equivalent field and press $\text{⌘}+\text{Shift}+\text{S}$, the standard keyboard shortcut for Save As. See Figure 7-1.

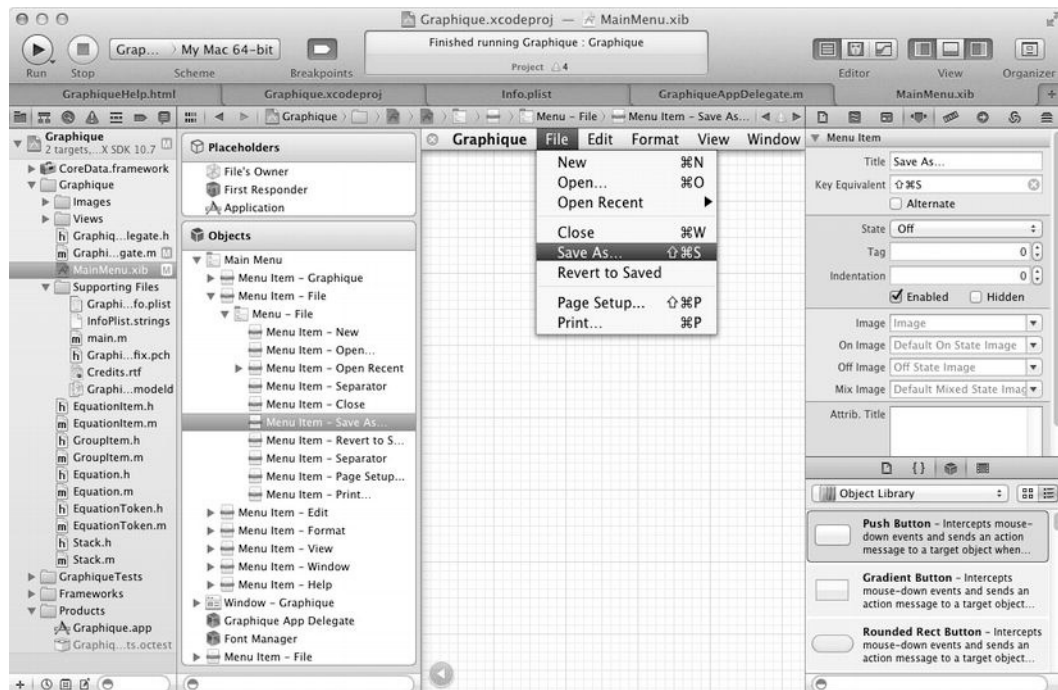


Figure 7-1. The Save... menu item changed to Save As...

The text now says Save As..., but if you open the Connections inspector, you see that the menu is linked to the saveDocument: method, not the saveDocumentAs: method. Delete the existing connection to saveDocument: by clicking the X next to First Responder (next to saveDocument:). Then, drag from the circle next to selector, under Sent Actions, to the First Responder object on the left. In the ensuing pop-up menu, select saveDocumentAs:, as shown in Figure 7–2.

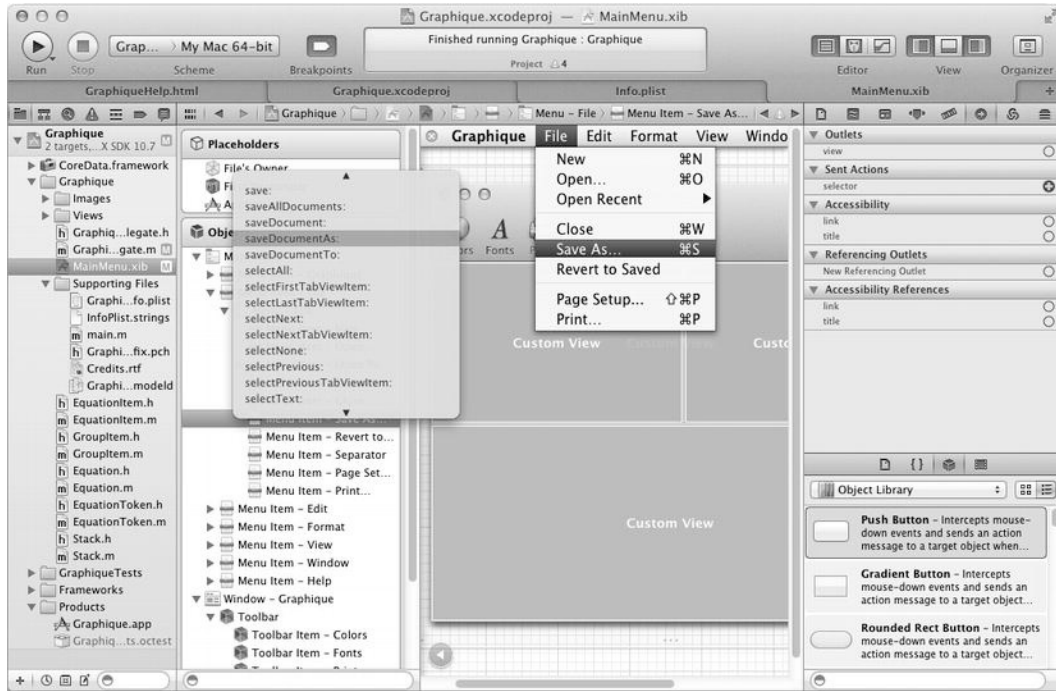


Figure 7–2. Connecting the Save As... menu item to the saveDocumentAs: method

When users select the Save As... menu item, the saveDocumentAs: action method will be sent up the responder chain. To respond, implement the saveDocumentAs: method in the first responder: the GraphiqueAppDelegate class. The implementation will perform the following steps:

1. Grab the current equation.
2. Prompt the user to select a location to save the new file.
3. Produce the property list file.
4. Save the file to the selected location.

Grabbing the Current Equation

Grabbing the current equation is as simple as querying the equation entry view controller for the current text in the text field, like this:

```
NSString *text = [self.equationEntryViewController.textField stringValue];
```

This line of code puts the text of the equation into the text variable. Since you don't actually need an Equation object, you leave it as a string.

Prompting the User to Select a Destination File

The `NSSavePanel` class prompts the user for a file location for saving the data. Listing 7-1 shows how this is done.

Listing 7-1. Opening an `NSSavePanel`

```
NSSavePanel *saveDlg = [NSSavePanel savePanel];
[saveDlg setAllowedFileTypes:[NSArray arrayWithObject:@"graphique"]];
NSInteger result = [saveDlg runModal];
if (result == NSOKButton) {
}
```

This code creates an `NSSavePanel` instance, constrains the allowed file types to `.graphique` files only (the standard Graphique file extension) and then calls the `runModal:` method to display the `NSSavePanel` instance as a modal window. It then tests the return value of the `runModal:` method to see whether the user clicked OK to see whether to actually save the file or cancel the operation.

Producing the Graphique File

As mentioned, Graphique files are simply property list files. To save the file, create an `NSDictionary` object with a single entry: `equation` as the key and the text in the text variable as the value. `NSDictionary` objects can conveniently be serialized into a property list file and can also be reread back into the object. Listing 7-2 shows the completed `saveDocumentAs:` method, which you should place in `GraphiqueAppDelegate.m`.

Listing 7-2. The `saveDocumentAs:` Method

```
- (void)saveDocumentAs:(id)sender
{
    // 1. Grab the current equation
    NSString *text = [self.equationEntryViewController.textField stringValue];

    // 2. Open the NSSavePanel
    NSSavePanel *saveDlg = [NSSavePanel savePanel];
    [saveDlg setAllowedFileTypes:[NSArray arrayWithObject:@"graphique"]];

    // Open the dialog and save if the user selected OK
    NSInteger result = [saveDlg runModal];
    if (result == NSOKButton){
        // 3. Producing the Graphique file
        NSMutableDictionary *data = [[NSMutableDictionary alloc] init];
        [data setObject:text forKey:@"equation"];
        [data writeToURL:saveDlg.URL atomically:YES];
    }
}
```

Launch the application and type an equation, as you normally would, in the equation entry field. Click the Graph button to view the graph so that you know it's something worth saving. Then, select **File** ► **Save As...** from the menu. This will open the NSSavePanel, as shown in Figure 7-3.

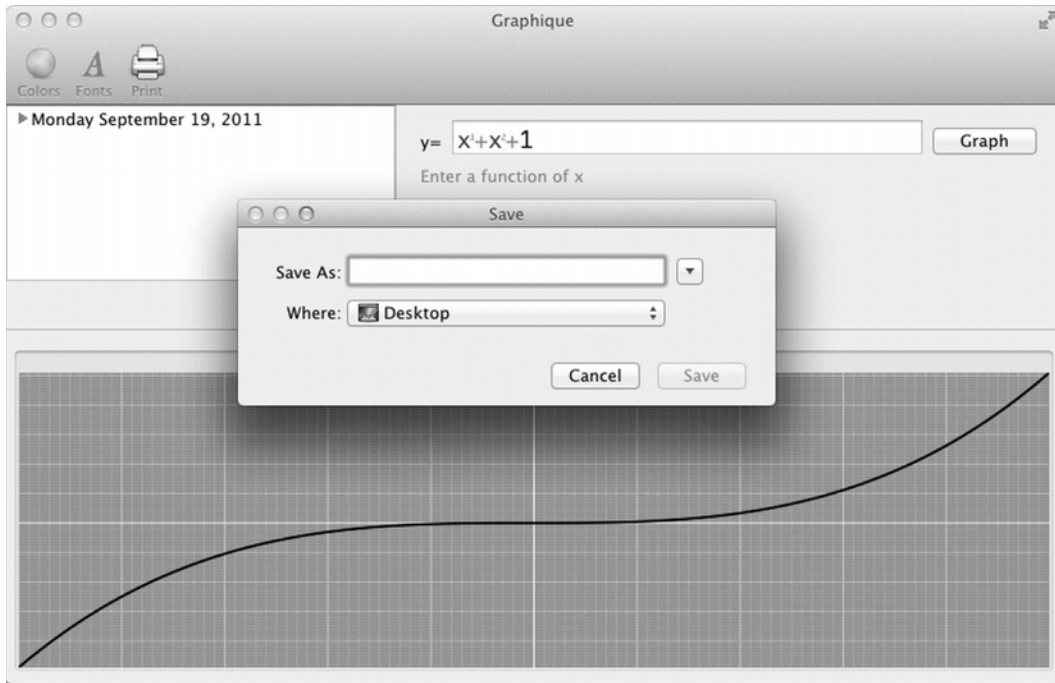


Figure 7-3. Prompting the user for a file location

Enter a file name and click the Save button. Finally, check the content of the newly created file using your favorite text editor. It should look something like the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>equation</key>
    <string>x3+x2+1</string>
</dict>
</plist>
```

As you can see, the dictionary object was serialized directly into the property list file. In fact, the Mac OS X property list files are backed by the NSDictionary class, which is why this file uses Apple's PropertyList DTD. Notice that the file is automatically given the .graphique extension because this is how you've configured the NSSavePanel. Graphique can now save equations to a file.

Loading a Graphique File into the Application

Now that Graphique can save equation files, it's only natural to provide a way to reopen the files at any time to view the equations again.

As we've shown in the previous section, you can use the MainMenu.xib file to find out what actions menu items are linked to. The Open menu item is automatically linked to a message called `openDocument:`. In `GraphiqueAppDelegate.m`, provide an implementation for this method, as shown in Listing 7-3. This implementation calls a new method that you haven't implemented yet, so initially it won't compile.

Listing 7-3. *The openDocument Method*

```
- (void)openDocument:(id)sender
{
    NSOpenPanel *openDlg = [NSOpenPanel openPanel];
    [openDlg setAllowedFileTypes:[NSArray arrayWithObject:@"graphique"]];

    NSInteger result = [openDlg runModal];
    if (result == NSOKButton)
    {
        NSMutableDictionary *data = [[NSMutableDictionary alloc]
initWithContentsOfURL:openDlg.URL];
        [self loadData:data];
    }
}
```

The method uses the counterpart to `NSSavePanel`, a class called `NSOpenPanel`, which allows users to open files. The code configures the panel to use Graphique files and lets the user choose the file to open. It then delegates the loading of the data to a `loadData` method that we still need to implement. We delegate to that method to have a central place where to input data into the application, regardless of how the data got into the application. In this case, the `NSOpenPanel` is the way in. Later, we learn how to load files by double-clicking them. At the end of `GraphiqueAppDelegate.h`, add a declaration for the `loadData` method:

```
- (void)loadData:(NSDictionary*)data;
```

Then implement the method in `GraphiqueAppDelegate.m` as shown in Listing 7-4.

Listing 7-4. *Loading the Graphique File*

```
- (void)loadData:(NSDictionary*)data
{
    NSString *equationText = [data objectForKey:@"equation"];
    Equation *equation = [[Equation alloc] initWithString:equationText];

    [self.equationEntryViewController.textField setStringValue:equation.text];
    [self.graphTableViewController draw:equation];

    [self.equationEntryViewController controlTextDidChange:nil];
}
```

The `loadData:` method uses a dictionary with the same format as the one we exported earlier, so all we need to do is produce a dictionary from the file and then extract the equation from it. `NSDictionary` provides a very convenient way to get to the data.

Start Graphique and select **File** ➤ **Open**. Choose the file you stored earlier. It loads the equation and plots it.

Graphique can now save and open Graphique files. Users expect to be able to double-click files in Mac OS X Finder to open them in the proper application, however. The next few sections show you how to launch Graphique when users double-click Graphique files from Finder, load the selected files, and display their equations.

Registering File Types with Lion

Any self-respecting operating system lets you register new file types so that users can double-click them and let the OS handle them. Some do it more elegantly than others. In Mac OS X, you have to reuse an existing or declare a new Uniform Type Identifier (UTI) to make the OS aware of file type associations. Apple defines a fairly large set of UTIs for common file formats such as PNG images or PDF documents. In our case, we need to create a new one for the `.graphique` extension.

NOTE: The list of default system-declared Uniform Type Identifiers is available from Apple at <http://developer.apple.com/library/mac/#documentation/Miscellaneous/Reference/UTIRef/Articles/System-DeclaredUniformTypeIdentifiers.html>.

You must perform three steps to properly handle the `.graphique` file type:

1. Define the new UTI for the `.graphique` extension.
2. Register the application as an editor for `.graphique` files.
3. Implement a method for handling the file.

The next sections walk through each of these steps in detail.

Defining the New UTI for the `.graphique` Extension

UTIs are declared in the application's `*-Info.plist` file under the `UTExportedTypeDeclarations` key. Xcode, however, gives us a simple interface for defining UTIs on the Info tab of the Targets view. To view it, select the root Graphique project in the Project navigator, make sure the Graphique target is selected in the middle of Xcode, and select the Info tab, as shown in Figure 7-4.

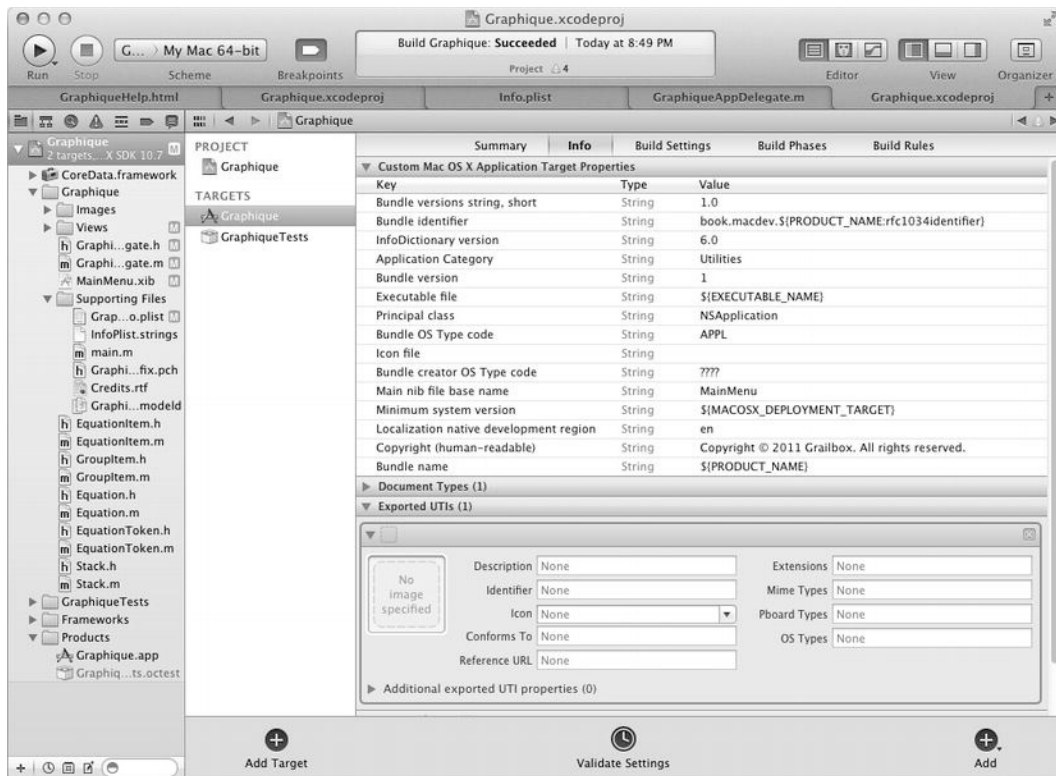


Figure 7–4. Viewing Info for the *Graphique* target

In there, we set some basic metadata, we set the file extensions we want to associate with the new UTI, and we give our type a unique identifier. Three fields are required:

- Identifier
- Conforms To
- Extensions

To these, we add a fourth, Description, just to make this UTI simpler to identify. Enter **Graphique Equation** in the Description field.

The Identifier field is what other components will use to when they want to refer to the new UTI. To ensure uniqueness, organizations typically use their inverted Internet domain identifier as a prefix for the type, usually matching the bundle identifier of the application. Often, they follow that with some basic name for the document type. For example, the UTI for PDF documents is `com.adobe.pdf`. The UTI for Excel spreadsheets in XLS format is `com.microsoft.excel.xls`. We'll stick with the same identifier as the bundle identifier; enter **book.macdev.graphique** in the Identifier field.

The Conforms To field declares what other UTIs the Exported Type UTIs being defined conform to, in a hierarchy similar to an object hierarchy. The Graphique file UTI should conform to two Apple-provided system UTIs: `public.content`, which is the base type for

all document content, and `public.data`, which provides the basis for byte stream data such as flat files and data on the clipboard. Enter **public.content**, **public.data** in the Conforms To field.

Finally, the Extensions field holds the valid file extensions for the UTI. Enter **graphique**, the only valid file extension for Graphique, in the Extensions field. The Exported UTIs section should now match Figure 7–5.

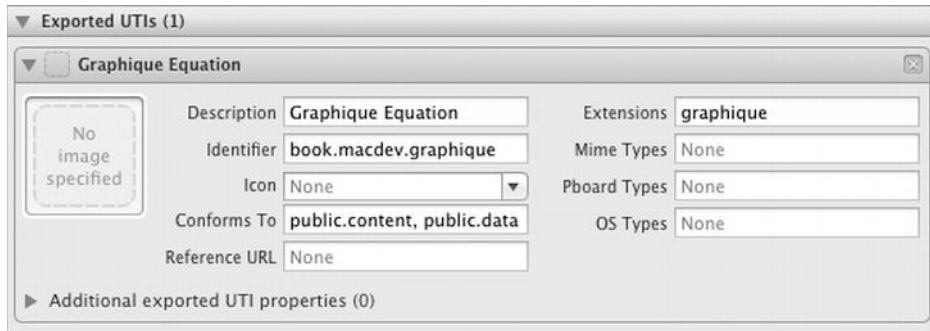


Figure 7–5. *The Exported UTIs configuration screen*

The Graphique application is now configured to export its UTI but solves only half the puzzle. You must also register Graphique as an editor for Graphique files. Read on to understand how to do that.

Registering Graphique as an Editor for Graphique Files

All we’ve done at this point is tell the operating system that `.graphie` files are “Graphique Equation” files. The next step is to advertise that Graphique is an application that can open Graphique Equation files. We configure this information on the same screen we configured the Exported UTIs section, in the Document Types section. Open that section to see what’s shown in Figure 7–6.



Figure 7–6. *The Document Types configuration screen*

Give this document type a human-readable name by typing **Graphique Equation** in the Name field. Enter the extension, **graphie**, in the Extensions field. Tie the document type to the exported UTI by entering **book.macdev.graphique** in the Identifier field.

Finally, select Editor in the Role field, indicating that Graphique can edit Graphique Equation files. The screen should match Figure 7-7.

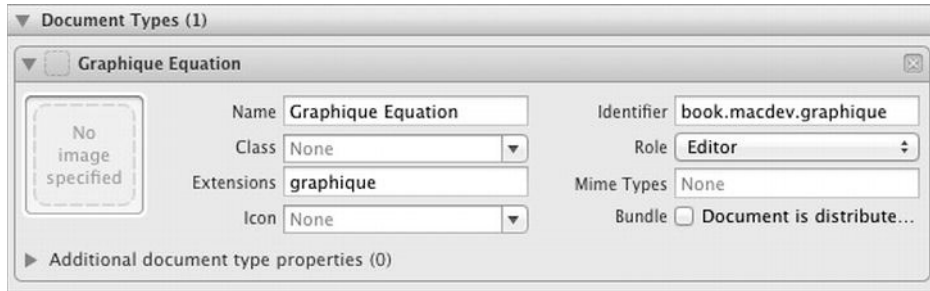


Figure 7-7. *The Graphique Equation document type*

Launch Graphique once to let the operating system know about the new exported UTI and document type. You can then close Graphique. Now, find a .graphie file you generated with the application earlier and Ctrl+click it. Select Open With, and see that Graphique is listed as an editor for it, as shown in Figure 7-8.



Figure 7-8. *The Graphique file type registered with the operating system*

Since you haven't yet implemented any method for opening files from Finder, trying to open the file fails. When users try to open a Graphique Equation file from Finder, you have two scenarios to handle: opening the file if Graphique isn't already running and opening the file if Graphique is already running. The next section implements code to handle both scenarios.

Handling Graphique Equation Files

When users open a file from Finder, the appropriate application is notified. More specifically, the `application:openFile:` method is called, passing in the name of the file that was opened. If Graphique is already running, it can just load the contents of the file and display the equation. If not, it can store the name of the file and then, when the `applicationDidFinishLaunching:` method is called after the application is ready to display its window, Graphique can load the contents of the file and display the equation. To store the name of the file, add a private attribute called `fileName` to the `GraphiqueAppDelegate.h` file, as shown in Listing 7–5.

Listing 7–5. *GraphiqueAppDelegate.h with a fileName Attribute*

```
#import <Cocoa/Cocoa.h>
#import <CoreData/CoreData.h>

@class EquationEntryViewController;
@class GraphTableViewController;
@class RecentlyUsedEquationsViewController;
@class PreferencesController;

@interface GraphiqueAppDelegate : NSObject <NSApplicationDelegate> {
    @private
    NSManagedObjectContext *managedObjectContext_;
    NSManagedObjectModel *managedObjectModel_;
    NSPersistentStoreCoordinator *persistentStoreCoordinator_;
    NSString *fileName;
}

@property (strong) IBOutlet NSWindow *window;
@property (weak) IBOutlet NSSplitView *horizontalSplitView;
@property (weak) IBOutlet NSSplitView *verticalSplitView;
@property (strong) EquationEntryViewController *equationEntryViewController;
@property (strong) GraphTableViewController *graphTableViewController;
@property (strong) RecentlyUsedEquationsViewController
*recentlyUsedEquationsViewController;
@property (strong) PreferencesController *preferencesController;

@property (nonatomic, readonly) NSManagedObjectContext *managedObjectContext;
@property (nonatomic, readonly) NSManagedObjectModel *managedObjectModel;
@property (nonatomic, readonly) NSPersistentStoreCoordinator
*persistentStoreCoordinator;

- (void)changeGraphLineColor:(id)sender;
- (IBAction)showPreferencesPanel:(id)sender;
- (void)loadData:(NSDictionary*)data;

@end
```

The `application:openFile:` method, implemented in `GraphiqueAppDelegate.m`, stores the name of the file in the `fileName` attribute. Then, if the application is already running, it loads the file using the `loadData:` method we’ve already implemented to load and display the equation. See Listing 7–6.

Listing 7-6. *Implementing application:openFile:*

```
- (BOOL)application:(NSApplication *)theApplication openFile:(NSString *)fileName_
{
    fileName = fileName_;

    if ([theApplication isRunning])
    {
        NSMutableDictionary *data = [[NSMutableDictionary alloc]
initWithContentsOfFile:fileName];
        [self loadData:data];
    }

    return YES;
}
```

We can then alter `applicationDidFinishLaunching:` in `GraphiqueAppDelegate.m` to add handling for opening the file, as shown in Listing 7-7.

Listing 7-7. *applicationDidFinishLaunching: with Handling for Opening a File Automatically*

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{
    self.equationEntryViewController = [[EquationEntryViewController alloc]
initWithNibName:@"EquationEntryViewController" bundle:nil];
    [self.verticalSplitView replaceSubview:[self.verticalSplitView subviews]
objectAtIndex:1] with:equationEntryViewController.view];

    self.graphTableViewController = [[GraphTableViewController alloc]
initWithNibName:@"GraphTableViewController" bundle:nil];
    [self.horizontalSplitView replaceSubview:[self.horizontalSplitView subviews]
objectAtIndex:1] with:graphTableViewController.view];

    self.recentlyUsedEquationsViewController = [[RecentlyUsedEquationsViewController
alloc] initWithNibName:@"RecentlyUsedEquationsViewController" bundle:nil];
    recentlyUsedEquationsViewController.managedObjectContext = self.managedObjectContext;
    [self.verticalSplitView replaceSubview:[self.verticalSplitView subviews]
objectAtIndex:0] with:recentlyUsedEquationsViewController.view];
    self.verticalSplitView.delegate = recentlyUsedEquationsViewController;

    [[NSColorPanel sharedColorPanel] setTarget:self];
    [[NSColorPanel sharedColorPanel] setAction:@selector(changeGraphLineColor:)];

    if (fileName != nil)
    {
        NSMutableDictionary *data = [[NSMutableDictionary alloc]
initWithContentsOfFile:fileName];
        [self loadData:data];
    }
}
```

Launch Graphique once to make sure the updates are sent to the operating system. Quit it immediately. Now you can double-click a `.graphique` file in Finder and watch the operating system launch Graphique and open the file. You can also double-click a `.graphique` file in Finder while Graphique is already running to display its equation.

Graphique can now respond when a user double-clicks a Graphique Equation file in Finder. Within Finder, however, Graphique Equation files look homely, with a bland, default icon. Read on to see how to improve the look of Graphique Equation files within Finder.

Using Quick Look to Generate Previews and Thumbnails

Mac OS X Leopard introduced the notion of file previews and thumbnails. When users select a file in Finder and press the spacebar, they see a preview of that file. If you do that with an image file, for example, you get a preview of the image. This technology is often referred to as “Quick Look”—you get a quick look at the file without actually opening an application to display it.

Also, Finder can display a thumbnail representing the contents of a file in place of a generic or an application-configured icon. Open a Finder window to your Pictures folder, for example, and look at the files Finder shows. For pictures or movies, it shows a thumbnail of the file, as shown in Figure 7–9.

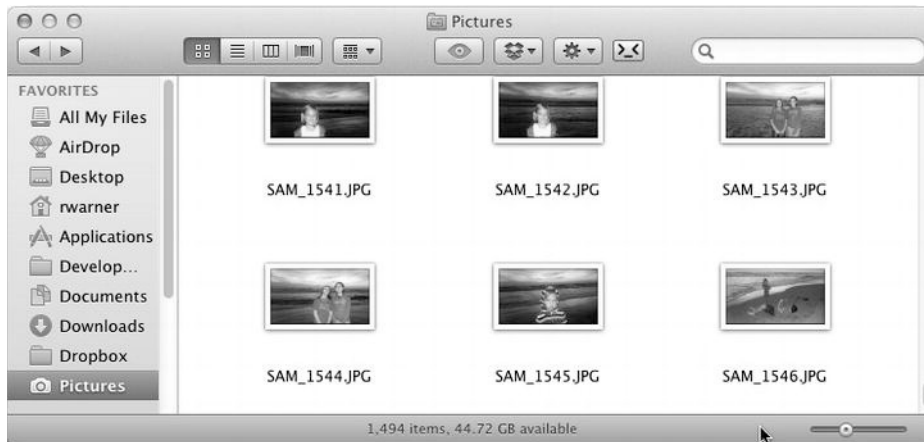


Figure 7–9. Pictures shown as thumbnails

In this section, you implement both previews and thumbnails for Graphique Equation files.

If you try to see a preview of a .graphique file by hitting the spacebar with it selected in Finder, all you get is an embarrassing reminder that we haven’t yet told the operating system how to handle previews for Graphique Equation files, as illustrated in Figure 7–10.

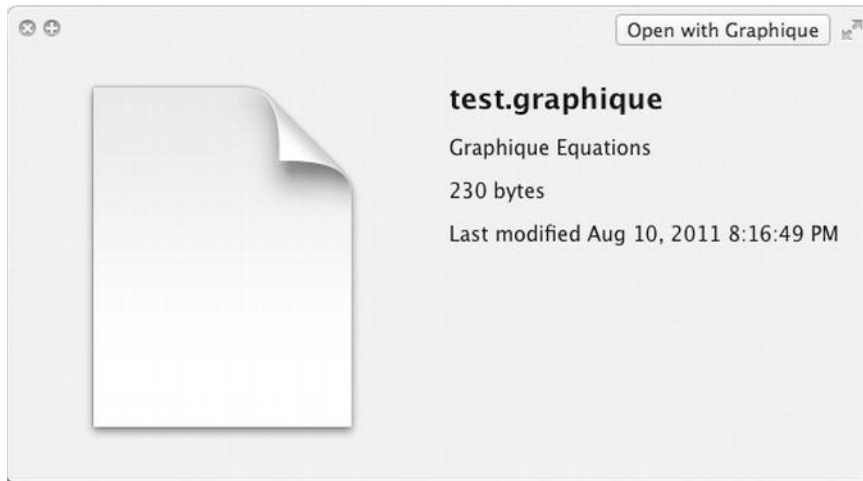


Figure 7–10. *Default preview*

Clearly, the blank document is not very professional and literally not portraying the right image of the application. Mac OS X uses a feature called Quick Look to produce previews that we can display instead of this blank document icon. A Quick Look plug-in is a binary that is deployed in the operating system that helps Finder generate previews using your own code. Ideally, we'd want the preview for an equation file to display a graph of the equation.

The same holds true for thumbnails. Thumbnails are smaller representations of the content of a file. They are used in the various Finder views. Once again, since the operating system doesn't know how to depict a Graphique file, it simply presents a blank generic document icon, as shown in Figure 7–11.

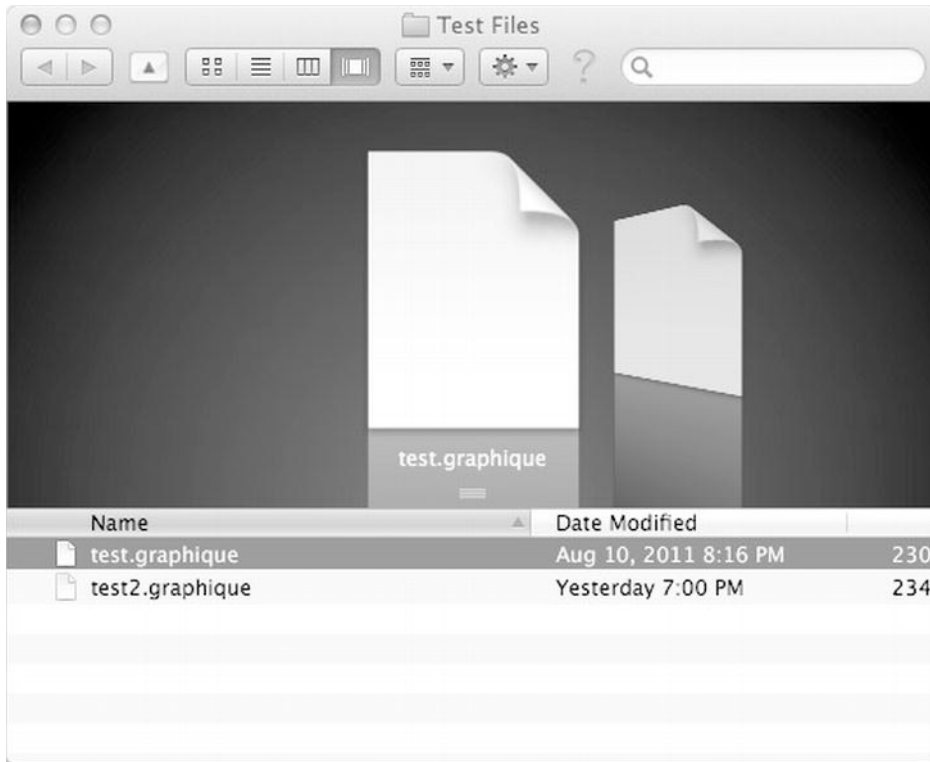


Figure 7-11. *Default thumbnail*

In this section, we show you how to implement a Quick Look preview and a thumbnail for .graphique files.

Creating the Quick Look Plug-in

A Quick Look plug-in produces an executable that is copied to /Library/QuickLook (or ~/Library/QuickLook for current user only installations). To create the plug-in, you create a new target within the Graphique project. Select the Graphique project node in the Graphique Xcode project, and choose **File** ► **New** ► **New Target** from the menu. In the list of targets, choose System Plug-In on the left and Quick Look Plug-In on the right, as shown in Figure 7-12, and click Next.



Figure 7-12. *Selecting the Quick Look Plug-In target*

The next screen prompts you for a product name. Enter **GraphiqueQL** for Product Name, enter **book.macdev** for Company Identifier, check the Use Automatic Reference Counting box, and make sure Graphique is selected for Project. Your screen should match Figure 7-13. Click Finish to generate the plug-in.

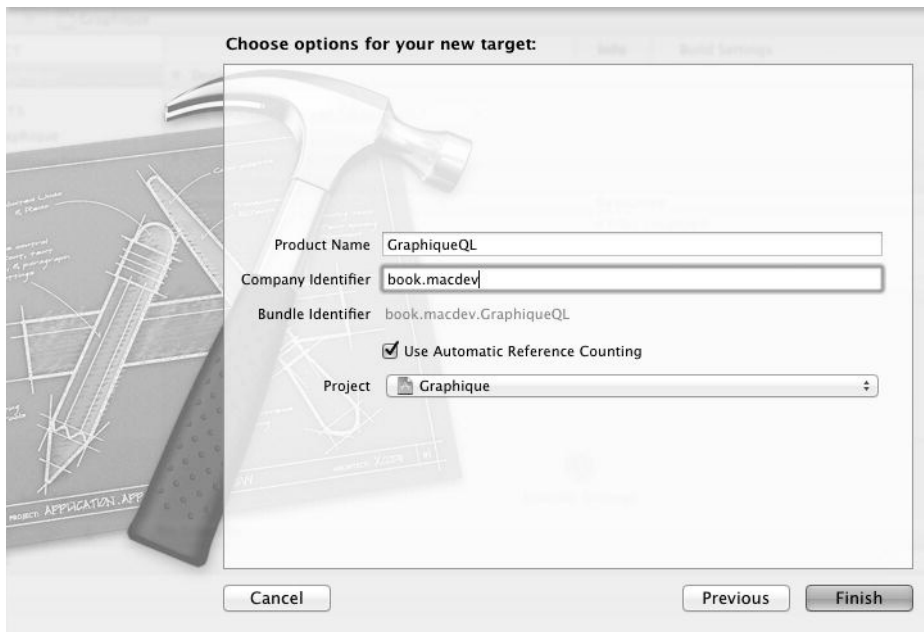


Figure 7-13. The options for the Graphique Quick Look plug-in

Xcode has generated the new plug-in with its related files in a group called GraphiqueQL, as illustrated in Figure 7-14.

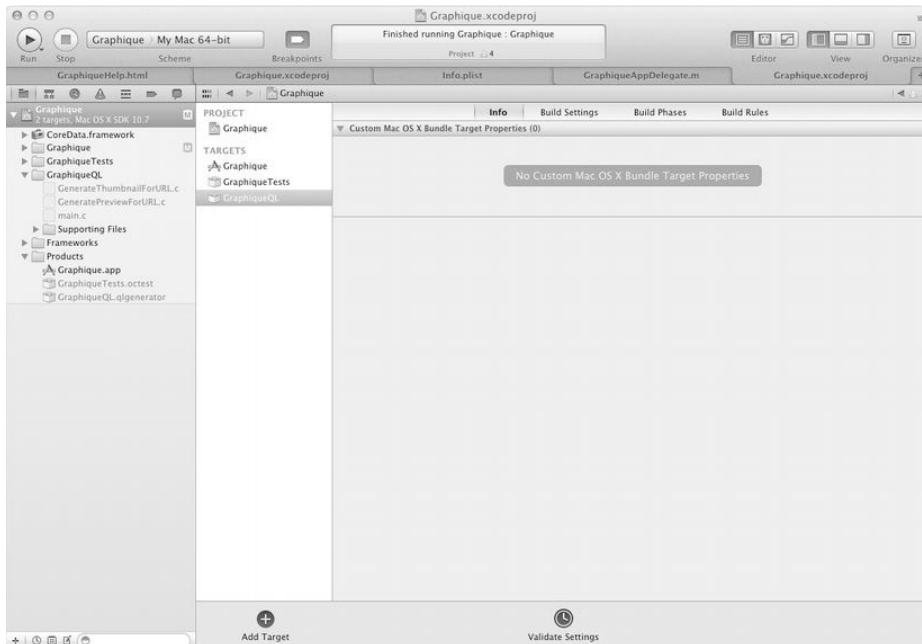


Figure 7-14. The Quick Look plug-in generated

The first thing to note is that the plug-in source code `GenerateThumbnailForURL` and `GeneratePreviewForURL` comes in `.c` files. Simply rename them with a `.m` extension so that you can use Objective-C syntax. Leave `main.c` with the `.c` extension.

You must tell the plug-in the type of file it supports, so select the `GraphiqueQL` target and the `Info` tab to see the associated property list. Note that you may have to select some other target and then reselect the `GraphiqueQL` target to coax it to display its contents. Expand the `Document types` section until you see the string `SUPPORTED_UTI_TYPE`. Change `SUPPORTED_UTI_TYPE` to **book.macdev.graphique**, as shown in Figure 7–15.

▼ Document types	Array	(1 item)
▼ Item 0	Diction...	(2 items)
▼ Document Content Type UTIs	Array	(1 item)
Item 0	String	book.macdev.graphique
Role	String	Quick Look Generator

Figure 7–15. Changing the Document Content Type UTIs

You’ve created the plug-in and told it what kind of file it supports, but your plug-in doesn’t actually generate previews or thumbnails yet. The next sections explain how to do that.

Implementing the Preview

As you might have guessed already, the code for generating the preview should be placed in `GeneratePreviewForURL.m`. This class bridges the Quick Look preview generator with your code.

You can follow one of two strategies for generating Quick Look previews:

- Save an image to use for the preview when you save the file from Graphique.
- Invoke the image generation code in the Quick Look preview each time a file is previewed.

Since we have an easy way to generate images from the view while Graphique is running, we opt for the first option. First, open `GraphTableViewController.h` and add the already-existing `export:` method to the public interface, as shown in Listing 7–8.

Listing 7–8. Adding the *export:* Method to the Public Interface

```
#import <Cocoa/Cocoa.h>
#import "Equation.h"

@class GraphView;

@interface GraphTableViewController : NSViewController <NSTableViewDataSource>

@property (nonatomic, retain) NSMutableArray *values;
@property (weak) IBOutlet NSTableView *graphTableView;
@property (nonatomic, assign) CGFloat interval;
```

```
@property (weak) IBOutlet GraphView *graphView;
@property (weak) IBOutlet NSTableView *tableView;

- (void)draw:(Equation *)equation;
- (NSBitmapImageRep *)export;
```

```
@end
```

Next, open `GraphiqueAppDelegate.m` and edit the `saveDocumentAs:` method, as shown in Listing 7–9, to create a bitmap image, in the PNG format, and store the image inside the Graphique Equation file.

Listing 7–9. *The `saveDocumentAs:` Method Stores a Preview in the File*

```
- (void)saveDocumentAs:(id)sender
{
    // 1. Grab the current equation
    NSString *text = [self.equationEntryViewController.textField stringValue];

    // 2. Open the NSSavePanel
    NSSavePanel *saveDlg = [NSSavePanel savePanel];
    [saveDlg setAllowedFileTypes:[NSArray arrayWithObject:@"graphique"]];

    // Open the dialog and save if the user selected OK
    NSInteger result = [saveDlg runModal];
    if (result == NSOKButton){
        // 3. Producing the Graphique file
        NSMutableDictionary *data = [[NSMutableDictionary alloc] init];
        [data setObject:text forKey:@"equation"];

        // Create the preview image
        NSBitmapImageRep *imageRep = [graphTableViewController export];
        NSData *img = [imageRep representationUsingType:NSPNGFileType properties:nil];
        [data setObject:img forKey:@"image"];

        [data writeToURL:saveDlg.URL atomically:YES];
    }
}
```

Open Graphique, type an equation, and save it into a `.graphique` file. If you open that file with a text editor, you will see that in addition to storing the equation, the file contains an image tag with the image data built in:

```
...
<dict>
    <key>equation</key>
    <string>x2+x3</string>
    <key>image</key>
    <data>
        iVBORwOKGgoAAAIHCAAAAA17UvFAAAKkm1DQ1BJQOMgUHJvZm1s
        ...
    </data>
</dict>
...
```

Generating the preview is now simply a matter of extracting this image data and painting it in the Quick Look graphics context. Open `GeneratePreviewForURL.m` and edit the `GeneratePreviewForURL` function to match Listing 7–10.

Listing 7–10. *The `GeneratePreviewForURL` Function*

```
OSStatus GeneratePreviewForURL(void *thisInterface, QLPreviewRequestRef preview,
CFURLRef url, CFStringRef contentTypeUTI, CFDictionaryRef options)
{
    NSRect canvas = {0, 0, 100, 100};

    CGContextRef cgContext = QLPreviewRequestCreateContext(preview, *(CGSize
*)&(canvas.size), false, NULL);
    if(cgContext) {
        NSGraphicsContext* context = [NSGraphicsContext
graphicsContextWithGraphicsPort:(void *)cgContext flipped:YES];
        if(context) {
            [NSGraphicsContext saveGraphicsState];
            [NSGraphicsContext setCurrentContext:context];

            NSDictionary *data = [[NSDictionary alloc] initWithContentsOfURL:(__bridge
NSURL*)url];
            NSData *imgData = [data objectForKey:@"image"];
            NSData *imgData = [data objectForKey:@"image"];
            [image drawInRect:canvas fromRect:NSZeroRect operation:NSCompositeSourceOver
fraction:1.0];

            [NSGraphicsContext restoreGraphicsState];
        }
        QLPreviewRequestFlushContext(preview, cgContext);
        CFRelease(cgContext);
    }

    return noErr;
}
```

You will need to make sure to import the Cocoa library at the top of the file:

```
#import <Cocoa/Cocoa.h>
```

Add `Cocoa.framework` to the list of frameworks that GraphiqueQL plug-in links to by selecting the Graphique project, the GraphiqueQL target, and the Build Phases tab. Expand the Link Binary With Libraries section, click the + button below it, and select `Cocoa.framework`. Xcode should look like Figure 7–16.

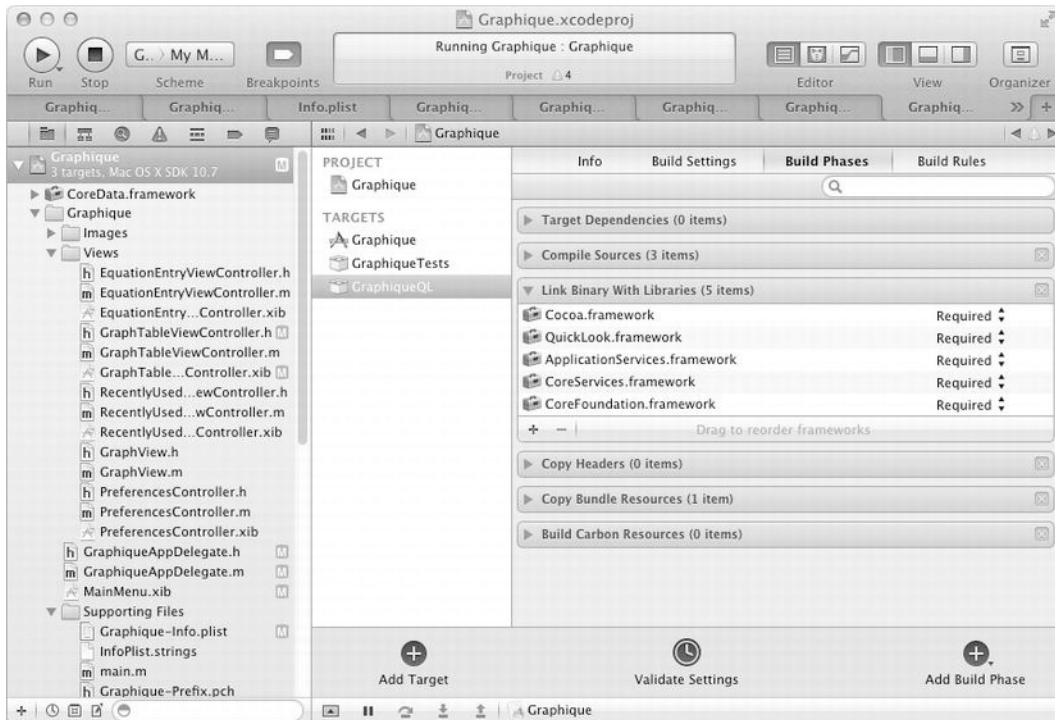


Figure 7-16. Adding *Cocoa.framework* to the libraries

Testing the Plug-in

Testing the plug-in requires a bit more setup. Quick Look comes with an executable called `qlmanage` located in `/usr/bin`. The steps required to set up Xcode 4 to run your Quick Look plug-in are detailed here.

Open a Terminal window and type the following:

```
cd ~ ; ln -s /usr/bin/qlmanage
```

You can then close the Terminal window. Go back to Xcode and do the following:

1. From the menu, choose **Product** ► **Manage Schemes** to list the current schemes.
2. Select **GraphiqueQL** and click the button labeled **Edit...**
3. Select the **Run** scheme on the left.
4. Select the **Info** tab.
5. From the Executable drop-down, select **Other...**

6. Select your home directory from the sidebar (Note: if your home directory isn't currently displayed in the sidebar, go into the Finder preferences, select the Sidebar tab, and check the box next to your home directory.)
7. Select the qlmanage link you created from the Terminal

The window should resemble Figure 7-17.

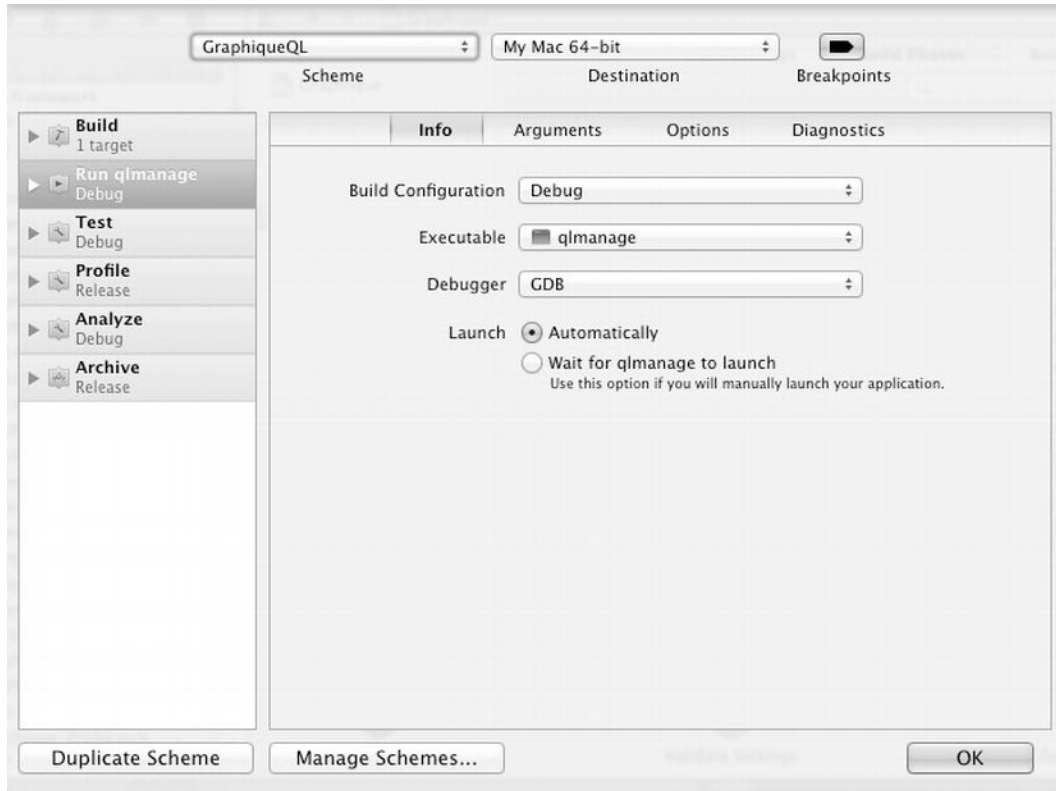


Figure 7-17. Setting up qlmanage as the QL plug-in executable

Next, go to the Arguments tab and, in the Arguments Passed on Launch section, add a -p argument with the path to your .graphique file (any such file you've created will work). For example, if the full path to your .graphique file were /Users/michael/Desktop/Test Files/test.graphique, you'd click the + button below the Arguments Passed On Launch section and type the following:

```
-p "/Users/michael/Desktop/Test Files/test.graphique"
```

See Figure 7-18.

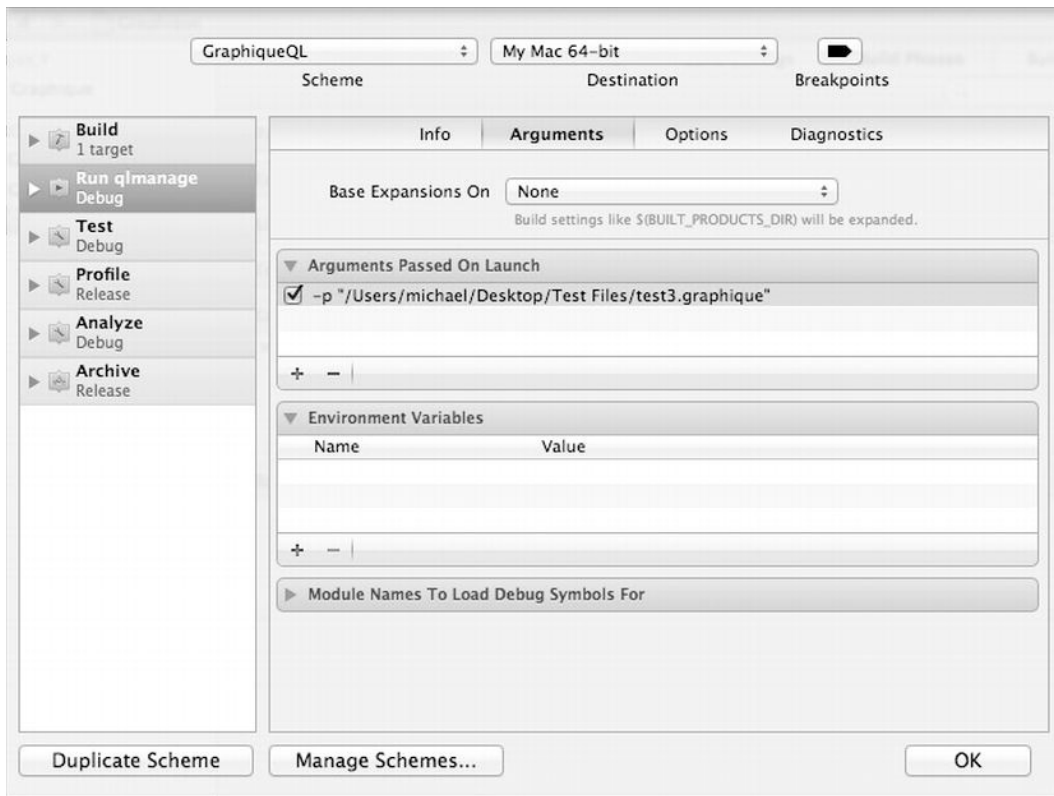


Figure 7-18. Specifying the test file to preview

Click OK to validate the run scheme and dismiss the dialog.

The last step of the setup process is to put the plug-in in a place where the operating system will find it. Follow these steps:

1. Select the Graphique project and select the GraphiqueQL target.
2. Go to the Build Phases tab and add a Copy Files build phase by clicking the Add Build Phase button in the bottom-right corner of Xcode and selecting Add Copy Files.
3. Expand the Copy Files sections it adds to the Build Phases.
4. For Destination, select Absolute Path.
5. For Subpath, enter `~/Library/QuickLook/`.
6. Click the + button at the bottom left of the selection.
7. Scroll to the bottom of the file list and select `GraphiqueQL.qllgenerator`, which you'll find beneath Products.

Your Xcode should match Figure 7-19.

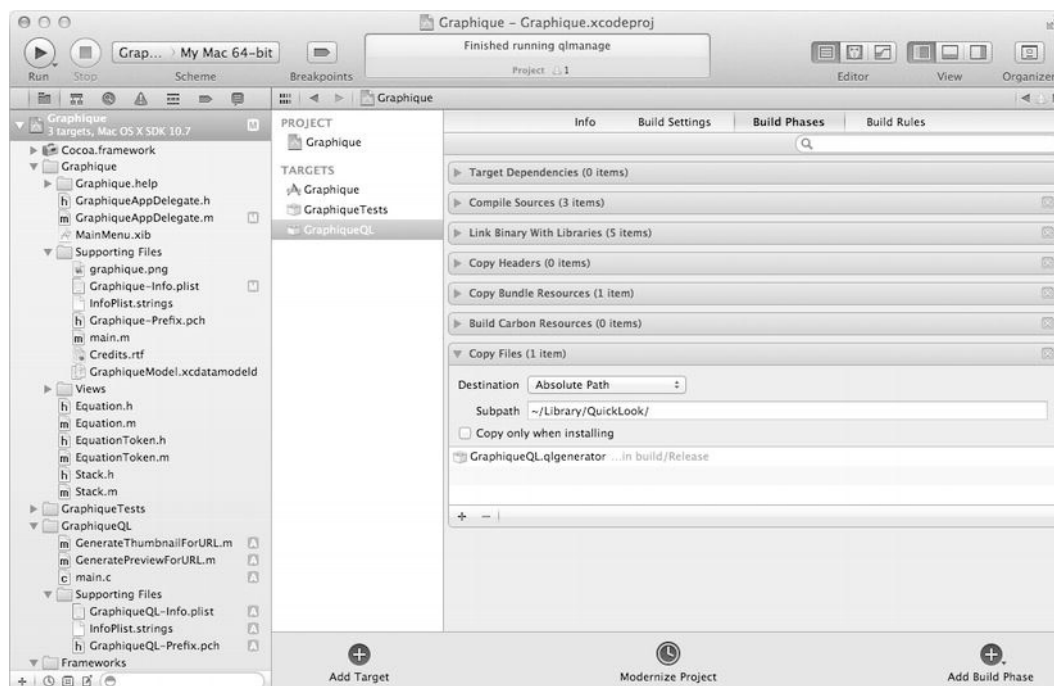


Figure 7-19. Configuring the QL plug-in build phases

Select the GraphiqueQL scheme, as illustrated in Figure 7-20, and hit the Run button to run the preview.



Figure 7-20. Selecting the GraphiqueQL scheme

If everything was configured properly, you should get a preview window like the one shown in Figure 7-21. Make sure the test file you are using was generated after amending the `saveDocumentAs:` method to save the preview in the file. If you don't have a test file that contains the preview image data, launch Graphique, create a test file, and return to this step.

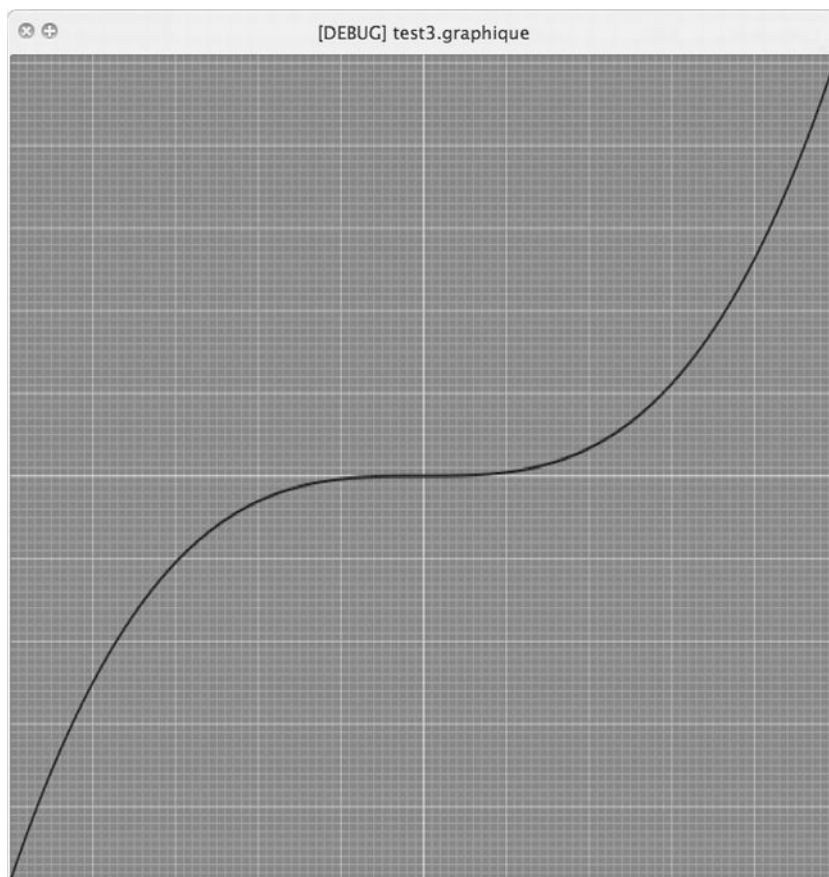


Figure 7–21. *A debug preview*

NOTE: You should now be able to see previews of your `.graphique` file from Finder. Quick Look uses a cache of known plug-ins. If it hasn't yet discovered your new plug-in and is still showing a generic preview, open a Terminal window and run the command `qlmanage -r` to make it reload its plug-ins.

Implementing the Thumbnail

Generating thumbnails is almost identical to generating previews. This time, we edit `GenerateThumbnailForURL.m` by adding an import for `<Cocoa/Cocoa.h>` at the top and implementing the `GenerateThumbnailForURL` function, as shown in Listing 7–11.

Listing 7–11. Generating a Quick Look Thumbnail

```

OSStatus GenerateThumbnailForURL(void *thisInterface, QLThumbnailRequestRef thumbnail,
CFURLRef url, CFStringRef contentTypeUTI, CFDictionaryRef options, CGSize maxSize)
{
    NSRect canvas = {0, 0, 100, 100};
    CGContextRef cgContext = QLThumbnailRequestCreateContext(thumbnail, *(CGSize
*)&(canvas.size), false, NULL);
    if(cgContext) {
        NSGraphicsContext* context = [NSGraphicsContext
graphicsContextWithGraphicsPort:(void *)cgContext flipped:YES];
        if(context) {
            [NSGraphicsContext saveGraphicsState];
            [NSGraphicsContext setCurrentContext:context];

            NSDictionary *data = [[NSDictionary alloc] initWithContentsOfURL:(__bridge
NSURL*)url];
            NSData *imgData = [data objectForKey:@"image"];
            NSData *imgData = [data objectForKey:@"image"];
            UIImage *image = [UIImage alloc] initWithData:imgData];
            [image drawInRect:canvas fromRect:NSZeroRect operation:NSCompositeSourceOver
fraction:1.0];

            [NSGraphicsContext restoreGraphicsState];
        }
        QLThumbnailRequestFlushContext(thumbnail, cgContext);
        CFRelease(cgContext);
    }

    return noErr;
}

```

You can test it in Xcode by editing the GraphiqueQL run schema, replacing the `qlmanage` argument from `-p` to `-t`, as shown in Figure 7–22.

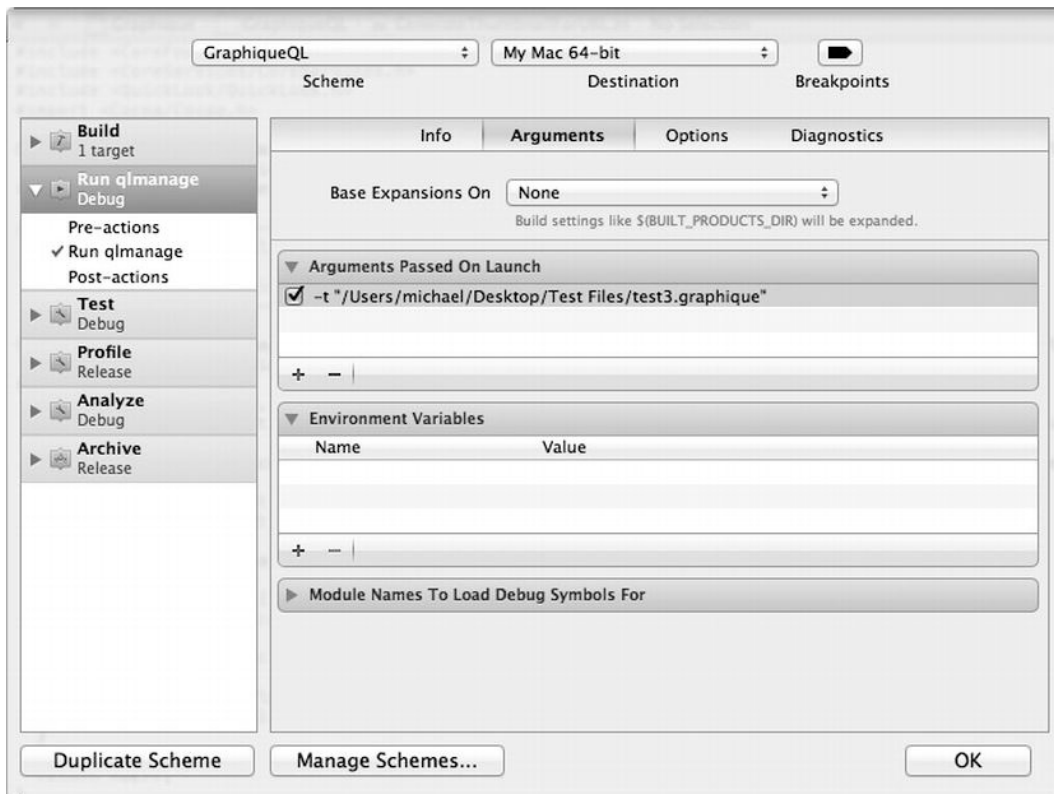


Figure 7-22. Changing the QL plug-in configuration to show thumbnails

Run GraphiqueQL in Xcode, and you should get a thumbnail image of your file, as illustrated in Figure 7-23.

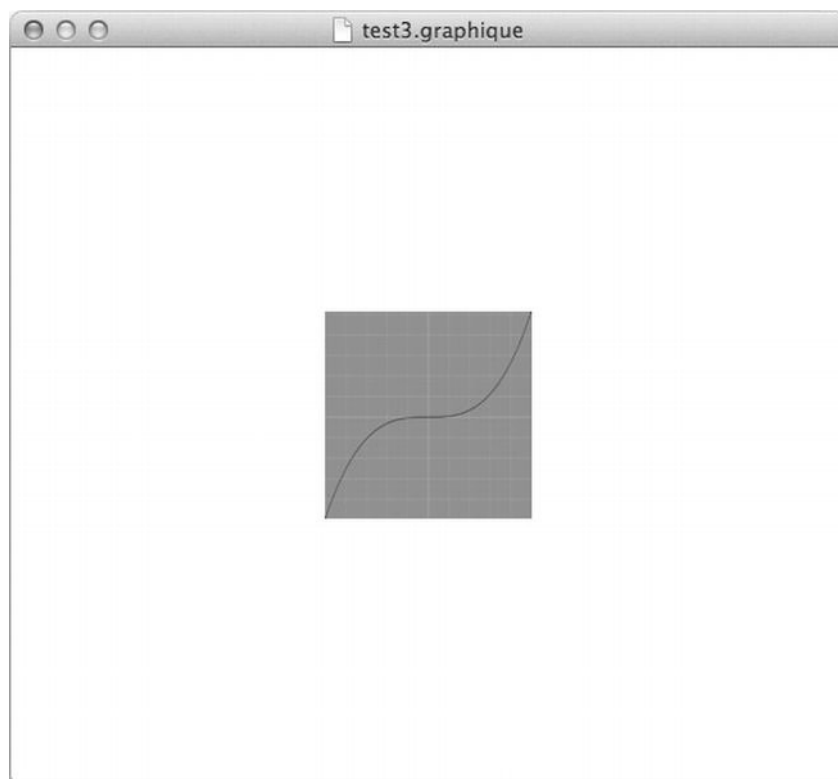


Figure 7–23. *Testing the Quick Look thumbnail*

Open a Terminal window and run `qlmanage -r` to reload the plug-ins. Then go to Finder and view your files to see the thumbnail and previews work. The Finder window should look like Figure 7–24.

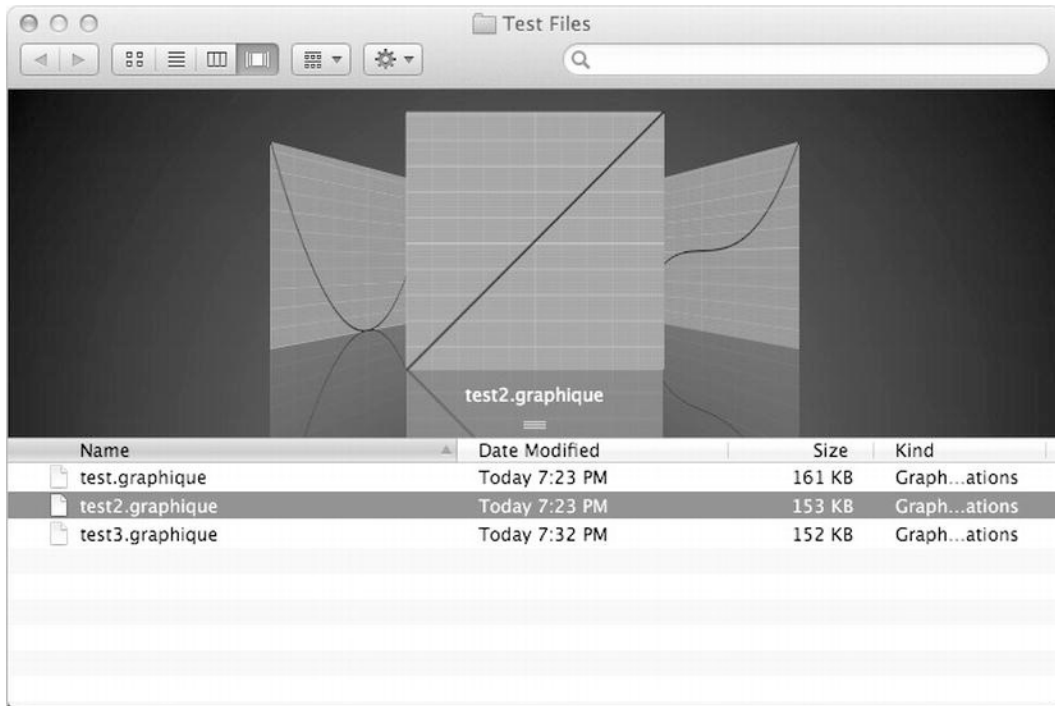


Figure 7–24. *Quick Look in action*

Distributing the Quick Look Plug-in with the Graphique Application

To finish your work on the Quick Look plug-in, you must set up the plug-in to be distributed with the Graphique application. First, dismantle the Copy Files Build Phase you created for testing the Quick Look plug-in by selecting the GraphiqueQL target and clicking the X in the upper-right corner of the Copy Files Build Phase you created.

Then, go to the Graphique target and, in the Target Dependencies section, add GraphiqueQL to the dependencies list, as shown in Figure 7–25. This will ensure that the Quick Look plug-in is built whenever Graphique is built.

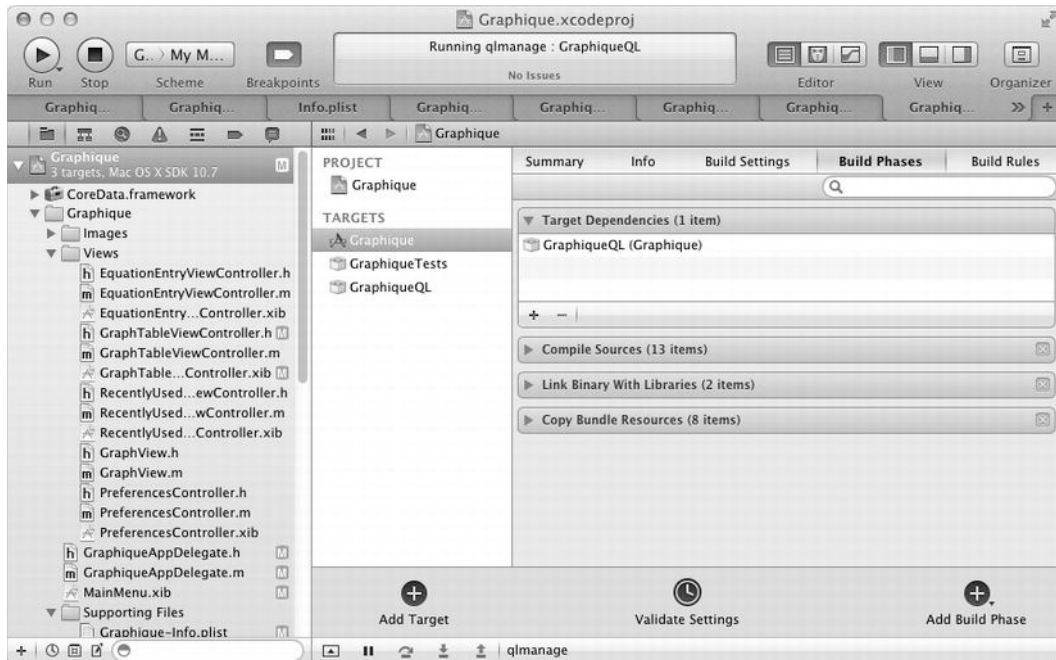


Figure 7–25. Adding *GraphiqueQL* as a dependency of *Graphique*

Quick Look plug-ins are stored in one of three locations:

- `/Library/QuickLook`
- `~/Library/QuickLook`
- `(Your application bundle)/Contents/Library/QuickLook`

Since the ultimate goal is to distribute Graphique from the Mac App Store, we want to put the Quick Look plug-in inside the Graphique application bundle. To accomplish this, we must copy the Quick Look plug-in into that directory.

Copying the Plug-in

To copy the Quick Look plug-in to the Graphique application bundle, click the Add Build Phase button and select Add Copy Files. Expand the Copy Files section that Xcode adds, select Wrapper in the Destination drop-down, enter **Contents/Library/QuickLook** in the Subpath field, and drag `GraphiqueQL.qlgenerator` from the list of files on the far left, under Products, and drop it into the Copy Files section.

The added Build Phase should match Figure 7–26.

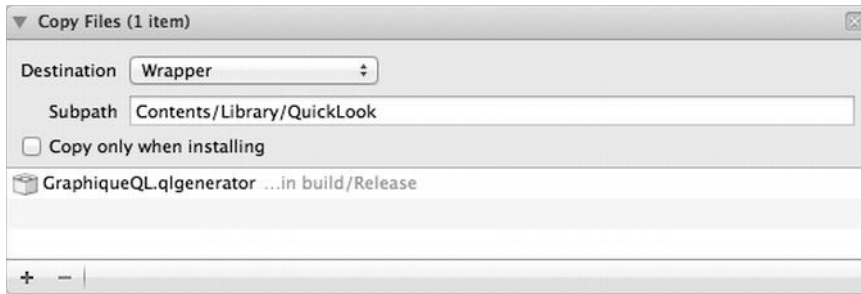


Figure 7–26. Setting up the Quick Look plug-in to copy into the application bundle

Now, when you build Graphique, the Quick Look plug-in will automatically be copied to the appropriate location within the application bundle.

Adding an Item to the Menu Bar

Across the top of screen, starting from the right corner and working left, Mac OS X displays icons in what it calls the *menu bar*. Some of these icons come standard with OS X, such as the ones for Spotlight or Volume Control. Others get put there by various third-party applications like Alfred or Twitter for Mac. Figure 7–27 displays a sample menu bar showing icons from both Mac OS X and from third-party applications.



Figure 7–27. Menu bar icons

These icons do various things, defined by the application that put them there. Some, like FaceTab or Blast, display the application’s only window hanging below the icon. Others, like Caffeine, incorporate the application’s entire user interface within the icon. Most, however, act like Twitter for Mac: they display a menu of actions so that you can quickly access application functions without having to switch to the application’s window. We’ll take that road for Graphique, creating a menu bar icon that, when clicked, displays the last ten recently used equations in a menu. Users will be able to click the Graphique menu bar item and then select an equation, and the application will display that equation and its corresponding graph or table.

Understanding NSStatusBar and NSStatusItem

Cocoa uses the `NSStatusBar` class to represent the menu bar and the `NSStatusItem` class to represent menu bar icons. The menu bar is a single, systemwide instance, which you can retrieve by calling `NSStatusBar`’s `systemStatusBar`: class method, like this:

```
NSStatusBar *statusBar = [NSStatusBar systemStatusBar];
```

You add icons to the system status bar using its `statusItemWithLength:` method, which not only creates the `NSStatusItem` instance but also returns it. The parameter you pass

to the `statusItemWithLength:` method is the desired width for the icon, in pixels. To create a status item that's 30 pixels wide, for example, you'd use code like this:

```
NSStatusBar *statusBar = [NSStatusBar systemStatusBar];  
NSStatusItem *statusItem = [statusBar statusItemWithLength:30.0];
```

Cocoa provides two constants that you can pass in place of the pixel width: `NSSquareStatusItemLength`, to make the width of the status item the same as its height, and `NSVariableStatusItemLength`, to tell the status bar to determine the width to fit its contents.

We've been saying "icon" each time we talk about status items, but you can also make status items display text (as the menu bar's clock does). To set a status item's text, call its `setTitle:` method. You can also create your own custom view for your status item and call `setView:` to use it. We'll stick with the traditional icon setup for Graphique, however, and call `setImage:`.

You control what happens when the user clicks your status item by either calling `setAction:`, passing a selector to be called any time the user clicks the status item, or creating a menu and setting the menu on the item using `setMenu:`. The menu then controls the action that's called when users click and navigate through it, just as any menu item does.

Since we now understand enough about status bars and status items, let's add the status item to Graphique.

Adding a Status Item to Graphique

Prepare for adding the status item by securing an icon for its use, either by creating one, downloading one, or using the one we've supplied with the source code. It should be a PNG file about 18 pixels square to fit with the other status bar icons. The one we've created is shown, zoomed in, in Figure 7-28.

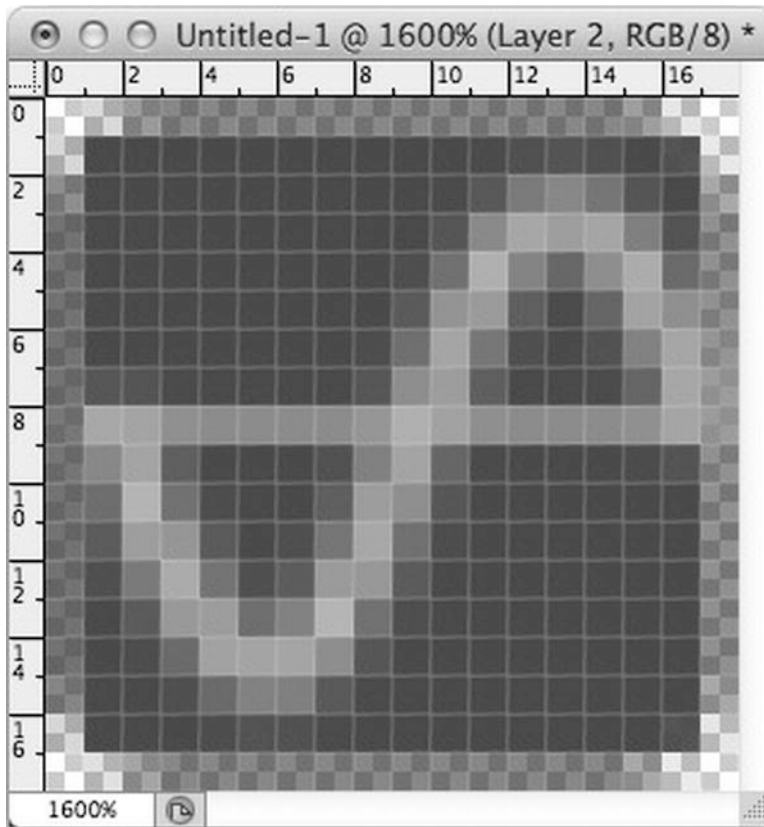


Figure 7–28. *The Graphique status item icon, zoomed in*

Name your icon `graphique18.png`, create a directory inside your Graphique project called `Images`, and save the icon file in that `Images` directory. Then, add the icon file to your project in Xcode, inside a new group called `Images`.

Building the Status Item Menu

Cocoa defines a protocol called `NSMenuDelegate` that offers methods for building menus. In this section, we create a class to implement this protocol so that we can use it from `GraphiqueAppDelegate` to handle menu chores. Create a new Cocoa class, derived from `NSObject`, called `GraphiqueStatusItemMenuDelegate`. This is the class that will build the menu and respond to selections on the status item's menu, so we make it conform to the `NSMenuDelegate` protocol. Listing 7–12 shows the code for `GraphiqueStatusItemMenuDelegate.h`.

Listing 7–12. *GraphiqueStatusItemMenuDelegate.h*

```
#import <Foundation/Foundation.h>

@class GroupItem;

@interface GraphiqueStatusItemMenuDelegate : NSObject <NSMenuDelegate>

@property (weak) GroupItem *rootItem;

- (id)initWithRootItem:(GroupItem *)rootItem;
- (void)statusMenuItemSelected:(id)sender;

@end
```

This class is initialized from a `GroupItem` instance, which you'll recognize from the Recently Used Equations view as representing a node in the tree. For our menu, we'll pass in the root node, so that we can access all equations in the tree.

This class also contains an action method, `statusMenuItemSelected:`, that will be called any time a menu item from the status item menu is selected.

You can create menus for status items in Interface Builder, just as you would with any other menu. If we were creating a static menu, with all the menu's items and actions known at build time, we'd most likely create our menu this way. We want to populate the menu in Graphique with the ten most recent equations, however, so we must build the menu dynamically. In fact, we really should build the menu each time the user clicks it, because the contents of the menu change any time the user graphs an equation. To have the opportunity to build the menu each time the user clicks it, we implement the `menuNeedsUpdate:` method, which gets called right before the menu is displayed.

To build the menu with the recent equations, we must work backward, first by date and then by the equations for each date, until we get to our limit of ten items. For each equation, we add a menu item with the equation as the text of the item. We use the same action for each menu item (`statusMenuItemSelected:`), and we pass an empty string for the shortcut key (passing `nil` for this parameter will crash the application). Listing 7–13 shows the implementation.

Listing 7–13. *GraphiqueStatusItemMenuDelegate.m*

```
#import "GraphiqueStatusItemMenuDelegate.h"
#import "GroupItem.h"
#import "Equation.h"
#import "GraphiqueAppDelegate.h"

#define MAX_ITEMS 10

@implementation GraphiqueStatusItemMenuDelegate

@synthesize rootItem;

- (id)initWithRootItem:(GroupItem *)rootItem_
{
    self = [super init];
```

```

    if (self != nil)
    {
        self.rootItem = rootItem_;
    }
    return self;
}

- (void)menuNeedsUpdate:(NSMenu *)menu
{
    // Remove all the menu items to rebuild from scratch
    [menu removeAllItems];

    // Keep track of how many menu items we've added so we know when we've reached the
    limit
    int numItems = 0;

    // Loop backwards through the date items
    for (NSInteger i = [rootItem numberOfChildren] - 1; i >= 0 && numItems < MAX_ITEMS; i--)
    {
        // For each date item, loop backward through the equations
        GroupItem *dateItem = [rootItem childAtIndex:i];
        for (NSInteger j = [dateItem numberOfChildren] - 1; j >= 0 && numItems < MAX_ITEMS; j--)
        {
            // For each equation, add a menu item with the equation as the menu title
            GroupItem *equationItem = [dateItem childAtIndex:j];
            [[menu addItemWithTitle:[equationItem text]
            action:@selector(statusMenuItemSelected:) keyEquivalent:@""] setTarget:self];

            // Increment the counter
            ++numItems;
        }
    }
}

- (void)statusMenuItemSelected:(id)sender
{
    // Get the selected menu item
    NSMenuItem *item = (NSMenuItem *)sender;

    // Graph the equation
    GraphiqueAppDelegate *delegate = NSApplication.sharedApplication.delegate;
    [delegate showEquationFromString:item.title];

    // Bring Graphique to the front
    [NSApp activateIgnoringOtherApps:YES];
}

@end

```

This code will not yet compile, because it expects a method called `showEquationFromString:` to exist in `GraphiqueAppDelegate`. The next section implements that method and performs the other steps necessary to integrate the status item menu.

Integrating the Status Item

To integrate the status item into the Graphique application, open `GraphiqueAppDelegate.h` and add members for the status item, the status item's menu, and the menu delegate. Also, add a method to set up the status item and to show an equation from a specified string. See Listing 7–14 for the updated `GraphiqueAppDelegate.h`.

Listing 7–14. *GraphiqueAppDelegate.h*

```
#import <Cocoa/Cocoa.h>
#import <CoreData/CoreData.h>

@class EquationEntryViewController;
@class GraphTableViewController;
@class RecentlyUsedEquationsViewController;
@class PreferencesController;
@class GraphiqueStatusItemMenuDelegate;

@interface GraphiqueAppDelegate : NSObject <NSApplicationDelegate> {
    @private
    NSManagedObjectContext *managedObjectContext_;
    NSManagedObjectModel *managedObjectModel_;
    NSPersistentStoreCoordinator *persistentStoreCoordinator_;
    NSString *fileName;

    NSStatusItem *statusItem;
    NSMenu *statusItemMenu;
    GraphiqueStatusItemMenuDelegate *statusItemMenuDelegate;
}

@property (strong) IBOutlet NSWindow *window;
@property (weak) IBOutlet NSSplitView *horizontalSplitView;
@property (weak) IBOutlet NSSplitView *verticalSplitView;
@property (strong) EquationEntryViewController *equationEntryViewController;
@property (strong) GraphTableViewController *graphTableViewController;
@property (strong) RecentlyUsedEquationsViewController
*recentlyUsedEquationsViewController;
@property (strong) PreferencesController *preferencesController;

@property (nonatomic, retain, readonly) NSManagedObjectContext *managedObjectContext;
@property (nonatomic, retain, readonly) NSManagedObjectModel *managedObjectModel;
@property (nonatomic, retain, readonly) NSPersistentStoreCoordinator
*persistentStoreCoordinator;

@property (strong) NSStatusItem *statusItem;
@property (strong) NSMenu *statusItemMenu;
@property (strong) GraphiqueStatusItemMenuDelegate *statusItemMenuDelegate;
```

```
- (void)changeGraphLineColor:(id)sender;
- (IBAction)showPreferencesPanel:(id)sender;
- (void)loadData:(NSDictionary*)data;
- (void)configureStatusItem;
- (void)showEquationFromString:(NSString *)text;
```

@end

Move on to `GraphiqueAppDelegate.m`. Add an import for the new delegate's header file:

```
#import "GraphiqueStatusItemMenuDelegate.h"
```

Add a `@synthesize` line for `statusItem`, `statusItemMenu`, and `statusMenuItemDelegate`:

```
@synthesize statusItem, statusItemMenu, statusMenuItemDelegate;
```

As the last line of the `applicationDidFinishLaunching:` method, add a call to the `configureStatusItem:` method that we're going to implement, like this:

```
[self configureStatusItem];
```

Listing 7–15 shows the implementation of `configureStatusItem:`. In that method, you create the status item, set its icon, and set it to highlight when selected. Then, you create the menu, leave it blank, and add it to the status item.

Listing 7–15. *The `configureStatusItem:` Method*

```
- (void)configureStatusItem
{
    // Create the status item
    self.statusItem = [[NSStatusBar systemStatusBar]
statusItemWithLength:NSVariableStatusItemLength];

    // Set the icon
    [statusItem setImage:[NSImage imageNamed:@"graphique18.png"]];

    // Set the item to highlight when clicked
    [statusItem setHighlightMode:YES];

    // Create the menu and delegate
    statusItemMenu = [[NSMenu alloc] init];
    self.statusItemMenuDelegate = [[GraphiqueStatusItemMenuDelegate alloc]
initWithRootItem:self.recentlyUsedEquationsViewController.rootItem];
    [statusItemMenu setDelegate:self];
    [statusItem setMenu:statusItemMenu];
}
```

Whoa—another compiler error. The `rootItem` member of `RecentlyUsedEquationsViewController` isn't available as a property, so add it in `RecentlyUsedEquationsViewController.h`:

```
@property (nonatomic, readonly) GroupItem *rootItem;
```

And in `RecentlyUsedEquationsViewController.m`, add this:

```
@synthesize rootItem;
```

Finally, you must implement the `showEquationFromString:` method in `GraphiqueAppDelegate.m`. This method should accept an equation as a string, create an equation from it, set its text into the equation entry field, and graph the equation. This sounds an awful lot like the existing `loadData:` method, except that it doesn't have to pull the equation string out of a dictionary. To leverage the existing code, refactor `loadData:` to pull the equation out of the specified dictionary, and then call the new `showEquationFromString:` method that now contains the rest of the code that was in `loadData:`. Listing 7–16 shows the refactored methods.

Listing 7–16. *The `loadData:` and `showEquationFromString:` Methods*

```
- (void)loadData:(NSDictionary*)data
{
    NSString *equationText = [data objectForKey:@"equation"];
    [self showEquationFromString:equationText];
}

- (void)showEquationFromString:(NSString *)text
{
    Equation *equation = [[Equation alloc] initWithString:text];

    [self.equationEntryViewController.textField setStringValue:equation.text];
    [self.graphTableViewController draw:equation];

    [self.equationEntryViewController controlTextDidChange:nil];
}
```

You can build and run Graphique now (make sure to switch the scheme back to Graphique if it's still set to GraphiqueQL) to see the status item added to the menu bar. You may need to graph a few equations to build your recently used pool, but then click the Graphique icon in the system menu. You should see a list of equations similar to Figure 7–29. Select one, and Graphique should graph it, coming to the front of your display if it wasn't already.

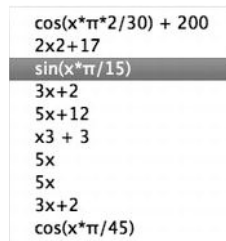


Figure 7–29. *The equations in the status menu*

Heeding Apple's Advice Regarding Menu Bar Icons

Apple's documentation for `NSStatusBar` warns against overuse of menu bar items, pointing out that space is limited and to create them only if other alternatives aren't appropriate. It highlights that the operating system doesn't guarantee that the menu bar icons will always be available (it might not have sufficient space to display them all) and

states that applications should always allow users to hide their menu bar icons. For Graphique, we add a user preference to hide the icon.

Adding the Preference to the Interface

To add this preference, we follow the pattern we established in Chapter 5 with the preference for which tab to display on Graphique's launch. We won't spend time explaining the steps, so if you have questions, refer to Chapter 5.

Start by opening `PreferencesController.h`, add a check box member for the preference, and add an action method for when the check box changes state. Listing 7-17 shows the updated `PreferencesController.h` file.

Listing 7-17. *PreferencesController.h*

```
#import <Cocoa/Cocoa.h>

@interface PreferencesController : NSWindowController
{
    NSButton *initialViewIsGraph;
    NSButton *showStatusItem;
}

@property (nonatomic, retain) IBOutlet NSButton *initialViewIsGraph;
@property (nonatomic, retain) IBOutlet NSButton *showStatusItem;

- (IBAction)changeInitialView:(id)sender;
- (IBAction)changeStatusItem:(id)sender;

@end
```

In `PreferencesController.m`, set the new check box in `windowDidLoad:` and add the implementation for the `changeStatusItem:` method to update the user defaults. In that method, we also must notify the application delegate that the check box changed so that it can either show or remove the status item. The compiler will complain after you add this code, because the `updateStatusItemState:` method doesn't yet exist on `GraphiqueAppDelegate`, but we'll soon rectify that. Listing 7-18 shows the updated `PreferencesController.m` file.

Listing 7-18. *PreferencesController.m*

```
#import "PreferencesController.h"
#import "GraphiqueAppDelegate.h"

@implementation PreferencesController

@synthesize initialViewIsGraph;
@synthesize showStatusItem;

- (id)init
{
    self = [super initWithWindowNibName:@"PreferencesController"];
    return self;
}
```

```

- (void)windowDidLoad
{
    [super windowDidLoad];

    // Get the user defaults
    NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

    // Set the checkbox to reflect the user defaults
    [initialViewIsGraph setState:[userDefaults boolForKey:@"InitialViewIsGraph"]];

    // Set the status item checkbox
    [showStatusItem setState:[userDefaults boolForKey:@"ShowStatusItem"]];
}

- (IBAction)changeInitialView:(id)sender
{
    // Get the user defaults
    NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

    // Set the user defaults value for the initial view
    [userDefaults setBool:[initialViewIsGraph state] forKey:@"InitialViewIsGraph"];
}

- (IBAction)changeStatusItem:(id)sender
{
    // Get the user defaults
    NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

    // Set the user defaults for the status item
    [userDefaults setBool:[showStatusItem state] forKey:@"ShowStatusItem"];

    // Notify the application delegate that the preference for the status item changed
    [(GraphiqueAppDelegate *)[NSApplication sharedApplication] delegate]
    updateStatusItemState];
}

@end

```

Open `PreferencesController.xib`, make the window bigger to accommodate another check box, and add the check box for setting the preference. Figure 7–30 shows the updated preferences window. Then, wire up the `showStatusItem` outlet and the `changeStatusItem:` action to the new check box.

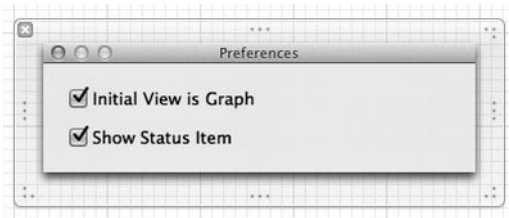


Figure 7–30. The preferences window with the status item preference

Now, switch to the `GraphiqueAppDelegate` class. In the header file (`GraphiqueAppDelegate.h`), declare the `updateStatusItemState:` method:

```
- (void)updateStatusItemState;
```

In `GraphiqueAppDelegate.m`, register the user default in the `initialize:` method to display the status item. Listing 7–19 shows the updated `initialize:` method.

Listing 7–19. *The initialize: Method*

```
+ (void)initialize
{
    // Get the user defaults
    NSUserDefaults *userDefaults = [NSUserDefaults standardUserDefaults];

    // Set the font to a reasonable choice and convert to an NSData object
    NSFont *equationFont = [NSFont systemFontOfSize:18.0];
    NSData *fontData = [NSArchiver archivedDataWithRootObject:equationFont];

    // Set the color to a reasonable choice and convert to an NSData object
    NSColor *lineColor = [NSColor blackColor];
    NSData *colorData = [NSArchiver archivedDataWithRootObject:lineColor];

    // Set the font, color, and status item in the defaults
    NSDictionary *appDefaults = [NSDictionary dictionaryWithObjectsAndKeys:fontData,
    @"equationFont", colorData, @"lineColor", [NSNumber numberWithInt:YES],
    @"ShowStatusItem", nil];
    [userDefaults registerDefaults:appDefaults];

    // Change the action for the Font Panel so that the text field doesn't swallow the
    changes
    [[NSFontManager sharedFontManager] setAction:@selector(changeEquationFont:)];

    // Set the color panel to show only Crayons mode
    [NSColorPanel setPickerMask:NSColorPanelCrayonModeMask];
}
```

The `updateStatusItemState:` method should look up whether to show the status item in the user defaults. If it should show the status item, it calls `configureStatusItem:`. If not, it removes the status item, releases the status item and menu, and sets them to `nil`. Listing 7–20 shows the `updateStatusItemState:` method.

Listing 7–20. *The updateStatusItemState: Method*

```
- (void)updateStatusItemState
{
    BOOL showStatusItem = [[NSUserDefaults standardUserDefaults]
    boolForKey:@"ShowStatusItem"];
    if (showStatusItem && statusItem == nil)
    {
        [self configureStatusItem];
    }
    else if (!showStatusItem && statusItem != nil)
    {
        [[NSStatusBar systemStatusBar] removeStatusItem:statusItem];
        statusItemMenu = nil;
        statusItem = nil;
    }
}
```

```
}  
}
```

Finally, change the call in `applicationDidFinishLaunching` from `configureStatusItem:` to `updateStatusItemState:` so it looks like this:

```
[self updateStatusItemState];
```

Now you can build and run Graphique, open the preferences window, and check and uncheck the Show Status Item check box to hide or display the Graphique status item.

Summary

John Donne said no man is an island, and we say no application should be one either. Successful applications integrate into the Mac OS X desktop, availing themselves of the services that Mac OS X offers. By integrating Graphique with the Finder's launcher, with Quick Look, and with the menu bar, you've increased Graphique's appeal and utility.

In a general sense, you should always seek ways to make your applications more integrated into the Mac OS X desktop. Users learn patterns of how applications work, and as they move from application to application, they expect their knowledge to transfer. By making your applications behave in ways that users expect, you help users get more use from your applications.

Creating Help

You've made your application as easy to use as possible. You've thought through the various workflows in your application and tweaked the user interface to make those flows intuitive and simple to understand. You've considered novices and experts and everyone in between, and you've crafted a UI masterpiece that should be eminently straightforward, discoverable, and useful.

And yet people don't always get it.

As much as you focus on improving your user interface (and, incidentally, that is indeed time well spent), your users likely will still need help navigating your screens and buttons and menus to accomplish the tasks they're using your application to fulfill. Help files can provide the handholding your users need to familiarize them with your application, discover new features, and perform tasks the way you've designed for them to be performed.

Although the trend established by iOS apps is to make your applications intuitive and discoverable to reduce the need for help, Mac OS X applications can still improve users' experiences by offering some guidance through help. If you run Graphique and select the **Help** ► **Graphique Help** menu item, however, you'll see the dialog shown in Figure 8–1. It's true: Graphique currently offers no help. In this chapter, we build help for Graphique.

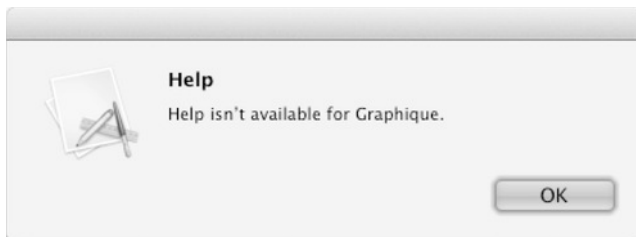


Figure 8–1. *Graphique offers no help.*

A Word on Help

Apple's documentation on help files can be confusing and outdated, admonishing you to include data that in fact isn't required, directing you to review complex examples as models for your help files (examples that break some of the rules in Apple's own documentation), and making you feel that the help on help is not at all helpful. Its documentation hasn't kept pace with the changes in new versions of the operating system, for whatever reason. Some apps have opted to ditch the confusion altogether and simply launch a browser and display their help from the Web, which is fine if your users are always connected to the Internet, have a fast connection, and don't have bandwidth cap concerns. Distributing help with your application is usually the better approach. This chapter clears up the mystery of help files, opting for a straightforward approach to crafting help that works.

Understanding Help Books

The collection of files that constitute the help that an application offers is called a *help book* and consists of HTML and XHTML files, a property list, graphics, QuickTime movies, and even AppleScript scripts. Apple recommends, for speed purposes, that the main file you create, which stands as the entry point to your help book, should conform to the XHTML 1.0 specification, while the rest of the help page files should conform to the HTML 4.01 specification. For the Graphique help book, we'll start by creating the main help file and then add a linked help page for one of the features. Within the text of the book, we won't reproduce the entire text of all the files in the completed help book to save space. You can peruse the finished product in the downloaded source. Instead, we'll focus on the essentials of help book creation so you can understand how to become a help author.

Like a Quick Look plug-in, your help book is a resource bundle that lives inside your application's resource bundle. When you're done adding your help book to Graphique, the resource bundle's directory structure will look like this:

```
| -Graphique.app
| ---Contents
| ----MacOS
| ----Resources
| -----Graphique.help
| -----Contents
| -----Resources
| -----English.lproj
| -----shared
| -----GraphiqueModel momd
| -----en.lproj
| ----Library
| -----QuickLook
| -----GraphiqueQL.qlgenerator
| -----Contents
| -----MacOS
| -----Resources
| -----en.lproj
```

Creating Your Help Book

You might be tempted to search for a Help Book template in Xcode’s “new file” templates, but such a search would end fruitlessly. Although help books have certain key elements and structures, you’re on your own for creating all the necessary files. Luckily, however, the files and structure are straightforward enough that you can re-create them. These are the steps to follow for creating a help book:

1. Create the directory structure for your resource bundle.
2. Create your main help file.
3. Create the rest of your help files, graphics, movies, and so on.
4. Create your help index.
5. Set up your plist file describing your help book.
6. Import your help book into your Xcode project.
7. Update your application’s plist file to point to the help book.
8. Set up Graphique’s build phases.

The order in which you perform these tasks generally doesn’t matter. As long as you do them all correctly, when you’re done, you can launch Graphique, select **Help ► Graphique Help** from the menu, and view help on Graphique.

Creating the Directory Structure

On your file system, inside the Graphique project directory, create a directory called `Graphique.help`. Your project directory should now have five subdirectories:

- `Graphique`
- `Graphique.help`
- `Graphique.xcodeproj`
- `GraphiqueQL`
- `GraphiqueTests`

If you view this directory in Finder, you’ll see that Finder, as with your application bundles with the `.app` extension, handles this directory differently from normal directories. It gives it a different icon—a life preserver, instead of a folder—and double-clicking it doesn’t open the folder in Finder but instead launches the help viewer and tries to display any help files in that bundle. This is good for launching help books but bad for navigating the directory structure while you’re creating the help book. To get around this, you can either launch a Terminal window and perform these tasks from the command line, or you can right-click the `Graphique.help` directory and select **Show Package Contents** to drill into the directory through Finder.

Inside the `Graphique.help` directory, create this directory structure:

```
| -Contents
| ---Resources
| -----English.lproj
| -----shared
```

Apple’s documentation suggests more directories than we show here, and indeed you can make the directory hierarchy as simple or as complex as you’d like. Smaller help books probably don’t merit complex directory structures, though, so we keep Graphique’s help book directory structure simple. As your project grows, however, you may find your help book easier to manage with more structure. Most of the directory names don’t matter. The ones that do are `Contents`, `Resources`, and `English.lproj`. See Table 8–1 for an explanation of each directory.

Table 8–1. *The Help Book Directories*

Directory Name	Purpose
Contents	Part of the bundle structure
Resources	Part of the bundle structure
English.lproj	The localized version of the help book for English. Each language you support will have its own localization directory
shared	Files that are the same across all localizations (typically graphics and style sheets)

Creating the Main Help File

In the `English.lproj` directory, create a file called `GraphiqueHelp.html`. When users select **Help ► Graphique Help** from the Graphique menu, this is the first help page they see. Typically, this file gives a brief description of the app and links to specific topics about the application. Follow suit by using the code shown in Listing 8–1.

Listing 8–1. *GraphiqueHelp.html*

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Graphique Help</title>
    <link rel="stylesheet" href="../shared/graphique.css" type="text/css"
media="screen">
  </head>
  <body>
    <h1>Graphique Help</h1>
    <p>Graphique is a graphing calculator that draws the equations you enter, both in
graphical and in tabular format.</p>
    <ul>
```

```

<li><a href="equations.html">Entering equations</a></li>
<li><a href="results.html">Seeing results</a></li>
<li><a href="recent.html">Reviewing recent graphs</a></li>
<li><a href="font.html">Changing the font</a></li>
<li><a href="color.html">Changing the graph's color</a></li>
<li><a href="status.html">Using the status item</a></li>
<li><a href="save.html">Saving a graph</a></li>
<li><a href="quicklook.html">Using QuickLook</a></li>
</ul>
</body>
</html>

```

You can see that this file has an XHTML 1.0 doctype and that the markup is simple. Open a browser and view this file, which you can easily do from the terminal by navigating to its directory and typing this:

```
open GraphiqueHelp.html
```

You should see a browser window that looks like Figure 8–2.

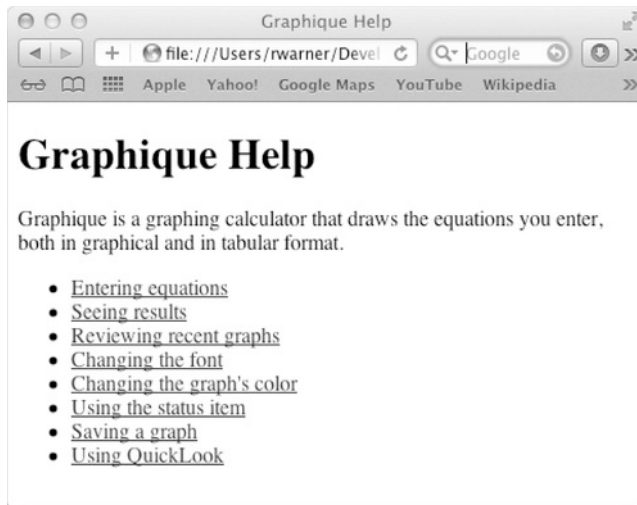


Figure 8–2. The main Graphique help file in a browser

This page references a style sheet called `graphique.css` in the shared directory. Create that style sheet, `shared/graphique.css`, to change the fonts used. See Listing 8–2 for the style sheet code.

Listing 8–2. *graphique.css*

```

body {
  background-color: #fff;
  font-family: "Lucida Grande", Helvetica, Arial, sans-serif;
  font-size: 14px;
  padding: 20px;
  margin: 0;
}

```

```
h1 {
  font-size: 1.2em;
  padding-left: 42px;
  background: url(../shared/graphique32.png) no-repeat top left;
  line-height: 32px;
}
```

You can see that the h1 style puts a 32x32 pixel icon to the left of its text. Create a 32x32 pixel icon, or copy it from the downloaded source, called `graphique32.png`, and place it in the shared directory alongside `graphique.css`. Refresh your browser window to see the effect of your styles. Figure 8–3 shows the updated help page.

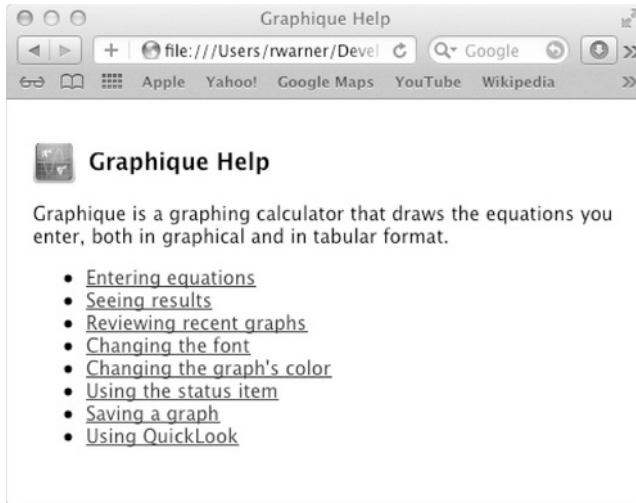


Figure 8–3. *The main Graphique help file after styling*

Creating the Rest of Your Help Files

This book walks you through creating one more help file—one that includes graphics. The additional help files follow the same pattern and are available in the downloaded source code for this book at www.apress.com/9781430237204. The file you'll create is the one for the first link on the main help page: `Entering Equations`. This means that none of the other links in your help book will work until you either create those pages or copy them in from source.

In the `English.lproj` directory, create a file called `equations.html` and edit it to match Listing 8–3.

Listing 8–3. *equations.html*

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
```



```

<link rel="stylesheet" href="../shared/graphique.css" type="text/css"
media="screen">
<title>Entering Equations</title>
<meta name="description" content="How to enter equations that Graphique can graph.">
<meta name="KEYWORDS" content="equation,trig">
</head>
<body>
<h1>Entering Equations</h1>
<p>You enter equations into Graphique's equation editor as a function of x. When no
equation is present, Graphique shows an example equation: 2*x+1.</p>

<p>Graphique supports addition, subtraction, multiplication, division, exponents,
and the trigonometric functions sin (sine) and cos (cosine).</p>
<p>You can depict exponents using the caret (^). Graphique also interprets as
exponents any numbers that follow x or a closed parenthesis.</p>
<p>Example:</p>

</body>
</html>

```

Two pieces of metadata merit explanation: description and KEYWORDS. The content for the description tag shows in search results below the name of the page. The values in KEYWORDS, separated by commas, offer additional items that will provide hits for this page in a search. You shouldn't include words that already appear in the page, because the search engine will already find those. You can see that we've included two keyword terms: a misspelling of *equation*, so that if the user mistypes the term in a search, this page will still be found. The other keyword that will find a hit on this page is *trig*.

This page references two graphics, shown in Figure 8–4 and Figure 8–5. Create these graphics from screenshots, or copy them from the downloaded source code, and put them in the shared directory.

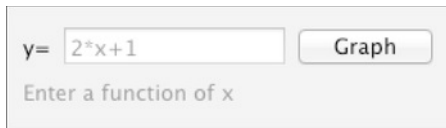


Figure 8–4. *equation_editor.png*

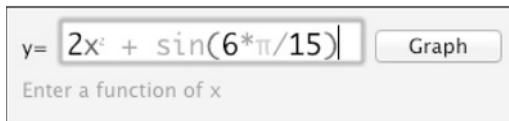


Figure 8–5. *equation_editor_example.png*

Go back to your browser window and click the Entering Equations link. Your browser window should now match Figure 8–6.

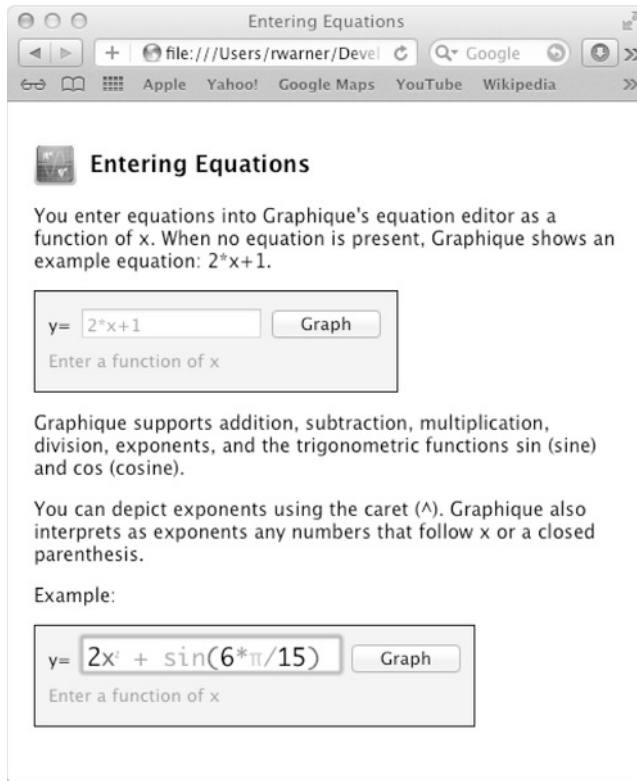


Figure 8–6. *The Entering Equations help page*

Before moving on, you need two more files: one, a 16x16 pixel icon that the help viewer uses for bookmarks, and the other, a file called `InfoPlist.strings`, which is used for localizing strings. Place the icon, called `graphique16.png`, in the shared directory. The `InfoPlist.strings` file goes in the `English.lproj` directory and has the content shown in Listing 8–4.

Listing 8–4. *The Entering Equations Help Page*

```
/* Localized versions of Info.plist keys */
HPDBBookTitle = "Graphique Help";
```

You can set this string to whatever you want to display in the help viewer's breadcrumb and search results.

Creating the Help Index

The help index allows users to search your help book and is a file you create and store in the localization folder of your help book (for example, `English.lproj`). You use a command-line utility called `hiutil` to create the index file. You can get more information on `hiutil` by viewing its man page. Any time you change the contents of your help book, you should re-create that file. You can also use a GUI app called `Help Indexer.app`, found in `/Developer/Applications/Utilities`, but its interface has quirks and serves simply as a front end for `hiutil` anyway.

Open a terminal and navigate to the `English.lproj` directory. Type the following command to create the help index:

```
hiutil -Cgf Graphique.helpindex .
```

When this command completes, verify that this command worked by using `hiutil` to list the files that your help index includes. At the prompt, type the following:

```
hiutil -Fvf Graphique.helpindex
```

You should see output like this:

```
/GraphiqueHelp.html
  Title: Graphique Help
  Descr: Graphique is a graphing calculator that draws the equations you enter, both
in graphical and in tabular format.

/equations.html
  Title: Entering Equations
  Descr: How to enter equations that Graphique can graph.
```

You see that both pages you've created are included in the index. You also see that the `-g` flag used when you created the help index generated a description for `GraphiqueHelp.html`, which you didn't explicitly specify in its HTML file. The `Entering Equations` page uses the description you explicitly specified.

Setting Up Your Plist File

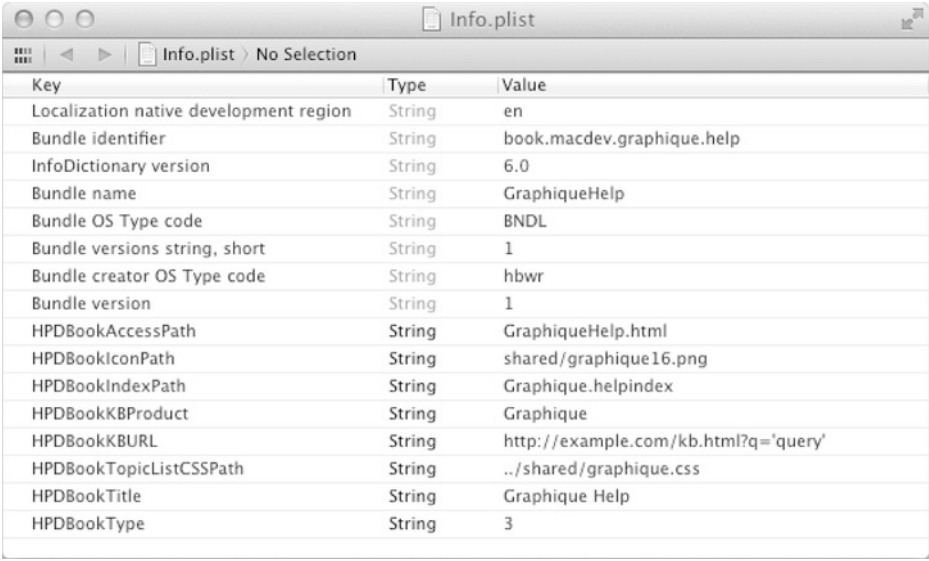
Your `Graphique.help` bundle must contain a plist file, called `Info.plist`, inside its `Contents` directory. This file contains keys that allow the Mac OS X help viewer to understand your help book and display it properly. Create the `Info.plist` file using Xcode, BBEdit, or some other plist-aware editor. If you're using Xcode, you create a new Property List file and then either you can open it as source and paste in the XML content or you can go through the Property List editor's interface and add the rows and values. If you're using BBEdit, you can just create a file called `Info.plist` and paste in the XML content. Listing 8-5 shows the XML content of `Info.plist`, and Figure 8-7 shows `Info.plist` in the Xcode editor.

Listing 8–5. Info.plist

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
  <dict>
    <key>CFBundleDevelopmentRegion</key>
    <string>en</string>
    <key>CFBundleIdentifier</key>
    <string>book.macdev.graphique.help</string>
    <key>CFBundleInfoDictionaryVersion</key>
    <string>6.0</string>
    <key>CFBundleName</key>
    <string>GraphiqueHelp</string>
    <key>CFBundlePackageType</key>
    <string>BNDL</string>
    <key>CFBundleShortVersionString</key>
    <string>1</string>
    <key>CFBundleSignature</key>
    <string>hbwr</string>
    <key>CFBundleVersion</key>
    <string>1</string>
    <key>HPDBookAccessPath</key>
    <string>GraphiqueHelp.html</string>
    <key>HPDBookIconPath</key>
    <string>shared/graphique16.png</string>
    <key>HPDBookIndexPath</key>
    <string>Graphique.helpindex</string>
    <key>HPDBookKBProduct</key>
    <string>Graphique</string>
    <key>HPDBookKBURL</key>
    <string>http://example.com/kb.html?q='query'</string>
    <key>HPDBookTopicListCSSPath</key>
    <string>../shared/graphique.css</string>
    <key>HPDBookTitle</key>
    <string>Graphique Help</string>
    <key>HPDBookType</key>
    <string>3</string>
  </dict>
</plist>

```



Key	Type	Value
Localization native development region	String	en
Bundle identifier	String	book.macdev.graphique.help
InfoDictionary version	String	6.0
Bundle name	String	GraphiqueHelp
Bundle OS Type code	String	BNDL
Bundle versions string, short	String	1
Bundle creator OS Type code	String	hbwr
Bundle version	String	1
HPDBookAccessPath	String	GraphiqueHelp.html
HPDBookIconPath	String	shared/graphique16.png
HPDBookIndexPath	String	Graphique.helpindex
HPDBookKBProduct	String	Graphique
HPDBookKBURL	String	http://example.com/kb.html?q='query'
HPDBookTopicListCSSPath	String	../shared/graphique.css
HPDBookTitle	String	Graphique Help
HPDBookType	String	3

Figure 8–7. *Info.plist* in Xcode's *plist* editor

See Table 8–2 for an explanation of each of the keys in *Info.plist*.

Table 8–2. *The Keys in Info.plist for Your Help Book*

Key	Value	Description
CFBundleDevelopmentRegion	en	Apple-supplied value
CFBundleIdentifier	book.macdev.graphique.help	The identifier for your help book; matches the value you'll put in your application bundle's <i>plist</i> file so your application can find your help book
CFBundleInfoDictionaryVersion	6.0	Apple-supplied value
CFBundleName	Name	Name for your help book bundle
CFBundlePackageType	BNDL	Apple-supplied value
CFBundleShortVersionString	1	Short version of your help book bundle
CFBundleSignature	hbwr	Apple-supplied value
CFBundleVersion	1	Version of your help book bundle

Key	Value	Description
HPDBookAccessPath	GraphiqueHelp.html	Your help book's title page
HPDBookIconPath	shared/graphique16.png	The icon to use for bookmarks
HPDBookIndexPath	Graphique.helpindex	The index file used for searches
HPDBookKBProduct	Graphique	Knowledge Base code to identify your application
HPDBookKBURL	http://example.com/kb.html?q='query'	URL to your support site
HPDBookTopicListCSSPath	../shared/graphique.css	Style sheet for topics
HPDBookTitle	Graphique Help	Title for help book
HPDBookType	3	Apple-supplied value

Importing Your Help Book into Your Xcode Project

In Xcode, Ctrl+click the Graphique folder and select **Add Files to “Graphique”...** to see the Open dialog for adding files. Select the Graphique.help directory, deselect the check box for “Copy items into destination group’s folder (if needed),” select the radio button for “Create folder references for any added folders,” and select the Graphique target. Click Add. Your Graphique.help directory should now appear in your project, as shown in Figure 8–8.

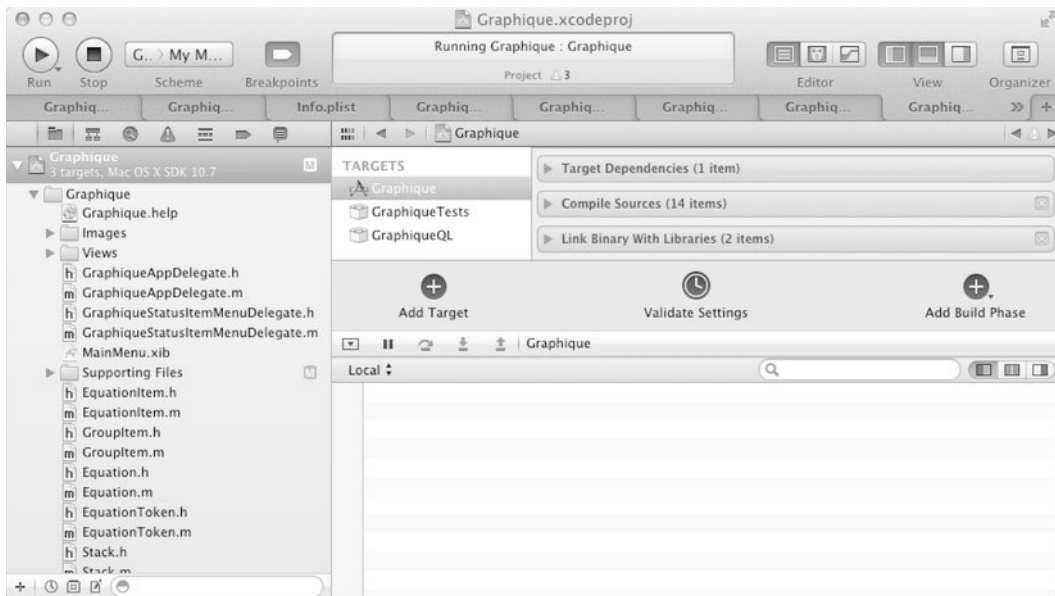


Figure 8–8. The help book files added to the Graphique project

Updating Your Application's Plist File

Your application must know a little more about your help book so that the help viewer can find it. To tie your help book to your application, add two keys to your application's plist file, `Graphique-Info.plist`. Listing 8–6 shows the code for the two keys, and Figure 8–9 shows the keys in the Xcode Property List editor.

Listing 8–6. Tying Your Help Book to Your Application

```
<key>CFBundleHelpBookFolder</key>
<string>Graphique.help</string>
<key>CFBundleHelpBookName</key>
<string>book.macdev.graphique.help</string>
```

Help Book directory name	String	Graphique.help
Help Book identifier	String	book.macdev.graphique.help

Figure 8–9. The Help Book Files Added to the Graphique project

Viewing the Help

Building and running Graphique registers your help book with the Mac OS X help viewer. You can now select **Help ► Graphique Help** to see your help book, as shown in Figure 8–10.

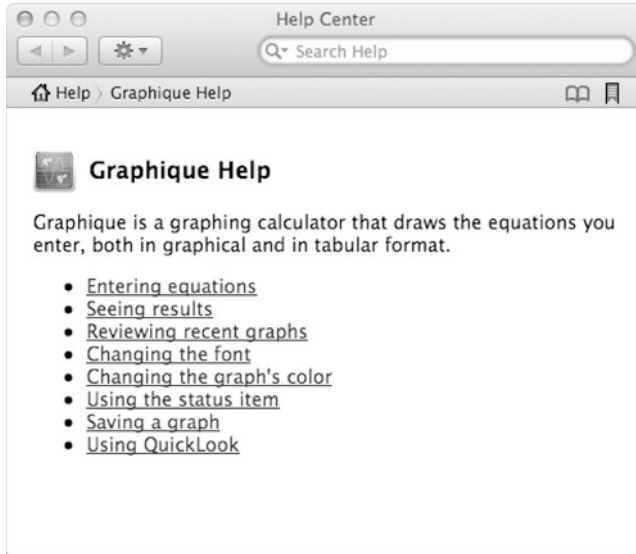


Figure 8–10. *The Graphique help book*

Click the Entering Equations link to verify that your subtopic help displays, as shown in Figure 8–11.

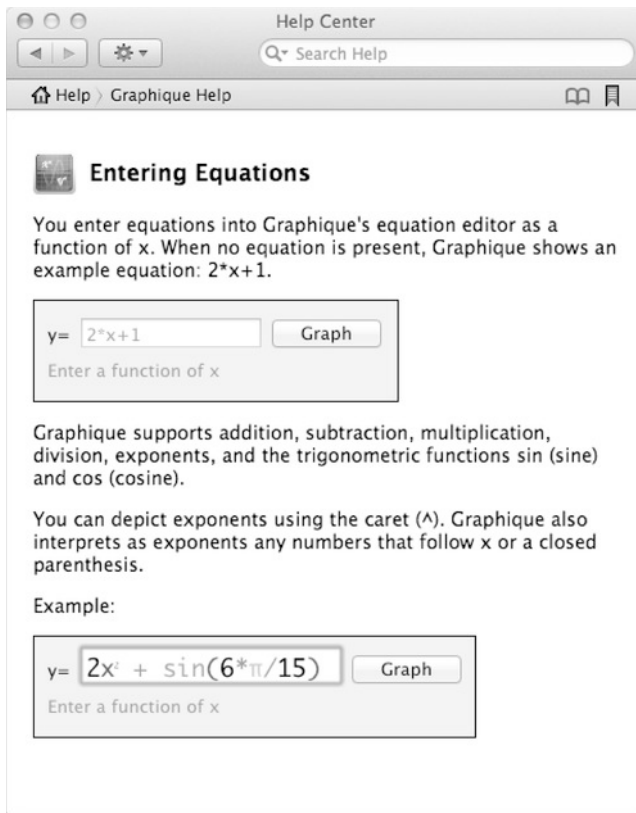


Figure 8–11. *The Entering Equations help*

Bookmarking a Page

With the Entering Equations page displayed, click the bookmark icon in the upper right of the help viewer window. This turns the icon red and, if it's your first bookmark, adds a book icon next to the bookmark. If you already had bookmarks, the book was already present. You can now click the bookmark icon to access all your bookmarks, as shown in Figure 8–12.

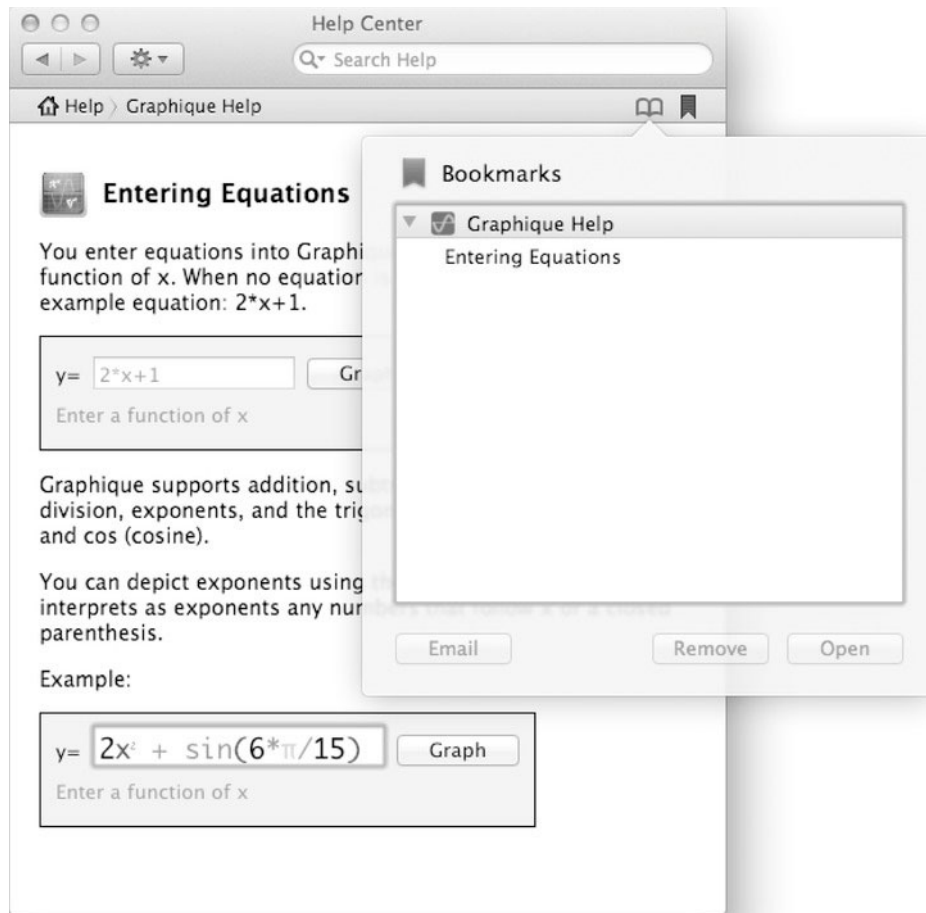


Figure 8–12. *Your bookmarks*

The icon displayed by the Graphique Help title is the 16x16 pixel icon you specified for the `HPDBookIconPath` key in `Info.plist`, `graphique16.png`. Selecting that bookmark will take you right to that help page, no matter where you are in the help viewer.

Performing a Search

The help viewer allows you to enter search terms in the search field, and it searches the current help book and all other help books on your system using their help index files. With the Graphique help showing, enter **equation** in the search field and press Enter. You should see both Graphique help pages listed, because they both contain the word *equation*. You also see the page descriptions listed below the page titles. Figure 8–13 shows the search results.

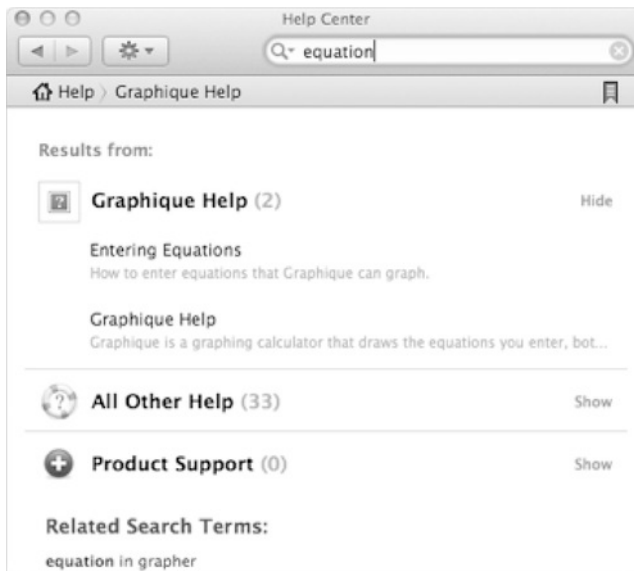


Figure 8–13. *The search results for equation*

Wait, the Graphique icon is missing! Actually, that's expected. The icon used for the search results comes from the application bundle, not the help book bundle. Because you haven't yet created an icon for Graphique, the help viewer has nothing to display. Revisit this screen in Chapter 10, after you've created the Graphique application icon.

Remember that you also included the misspelled *eqaution* as a keyword for the Entering Equations page. Try entering **eqaution** in the search field now, and you'll see that the help viewer does indeed find your Entering Equations page, as Figure 8–14 shows.

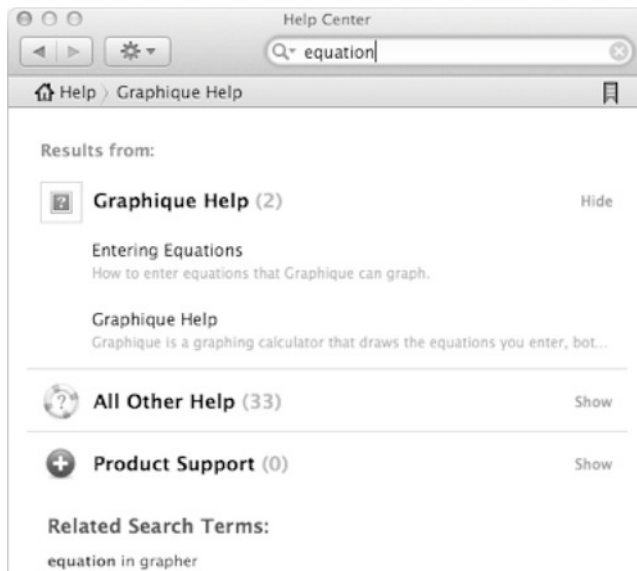


Figure 8–14. *The search results for the misspelled eqaution*

Summary

Software engineers often claim that well-written and intuitive software does not need any documentation. Although this may be true in the eyes of a technically savvy individual, there are legions of users out there who don't adhere to this pernicious philosophy and are silently internalizing their frustration while trying to figure out how to use an app. The App Store gives them an outlet to voice their accumulated frustration and take a shot at your rankings and profits. In a marketplace environment like the App Store, the ratings you get are paramount to your success. Anything you do to keep the users from getting frustrated with the app and leaving a bad review in the App Store will translate into higher rankings.

Printing

Although we live in the digital age, most applications that produce data offer support for printing that data. Word processors obviously allow users to print their documents, but even the Mac OS X Calculator has an option for printing its virtual paper tape. Users expect to preserve the words they type, the numbers they enter, and the graphics they create onto paper and ink. In this chapter, we explore some of the printing capabilities of Mac OS X and how to add printing support to Graphique. We start with a naïve printing implementation of the graph view that would use enough ink to put Hewlett-Packard back in the black. Next, we optimize the graph view for printing so that we can feel good about actually printing it. Finally, we build a print view that spans three pages and includes both the graph view and the table view.

Printing the Graph View

With Cocoa, Apple offers a printing framework that is simple to understand and execute. The `NSView` class exposes a `print:` method that requests that the view draw itself and its subviews onto a printer device. We use this method in our very first, simplistic approach to printing the `GraphView`.

Printing the Graph

As usual, we continue with the Graphique project. Run Graphique, type an equation, and hit the Graph button. Your equation will be graphed as usual. Leave the focus on the equation entry field. If you then choose **File** ► **Print** from the Graphique menu, the print preview window opens, and the preview shows the equation you typed, in rather small type, as shown in Figure 9–1.

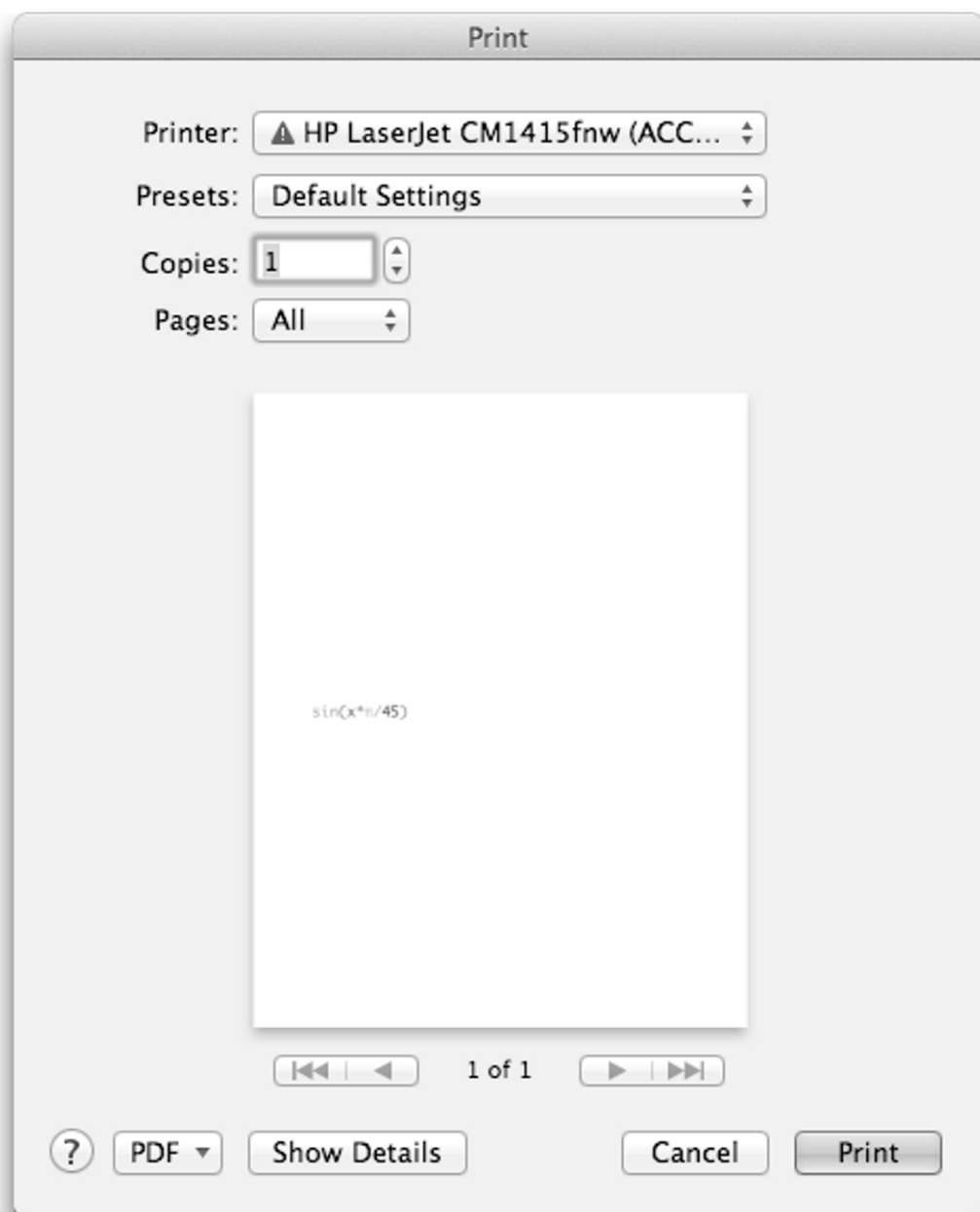


Figure 9–1. The print preview window showing the equation

You are witnessing the default print behavior of printing the first responder view, which is the equation entry field since it has the focus. If you click the Recently Used Equations

view, however, and then select **File** ► **Print** from the menu, you instead see the tree view of recent equations in the print preview window, as shown in Figure 9–2.

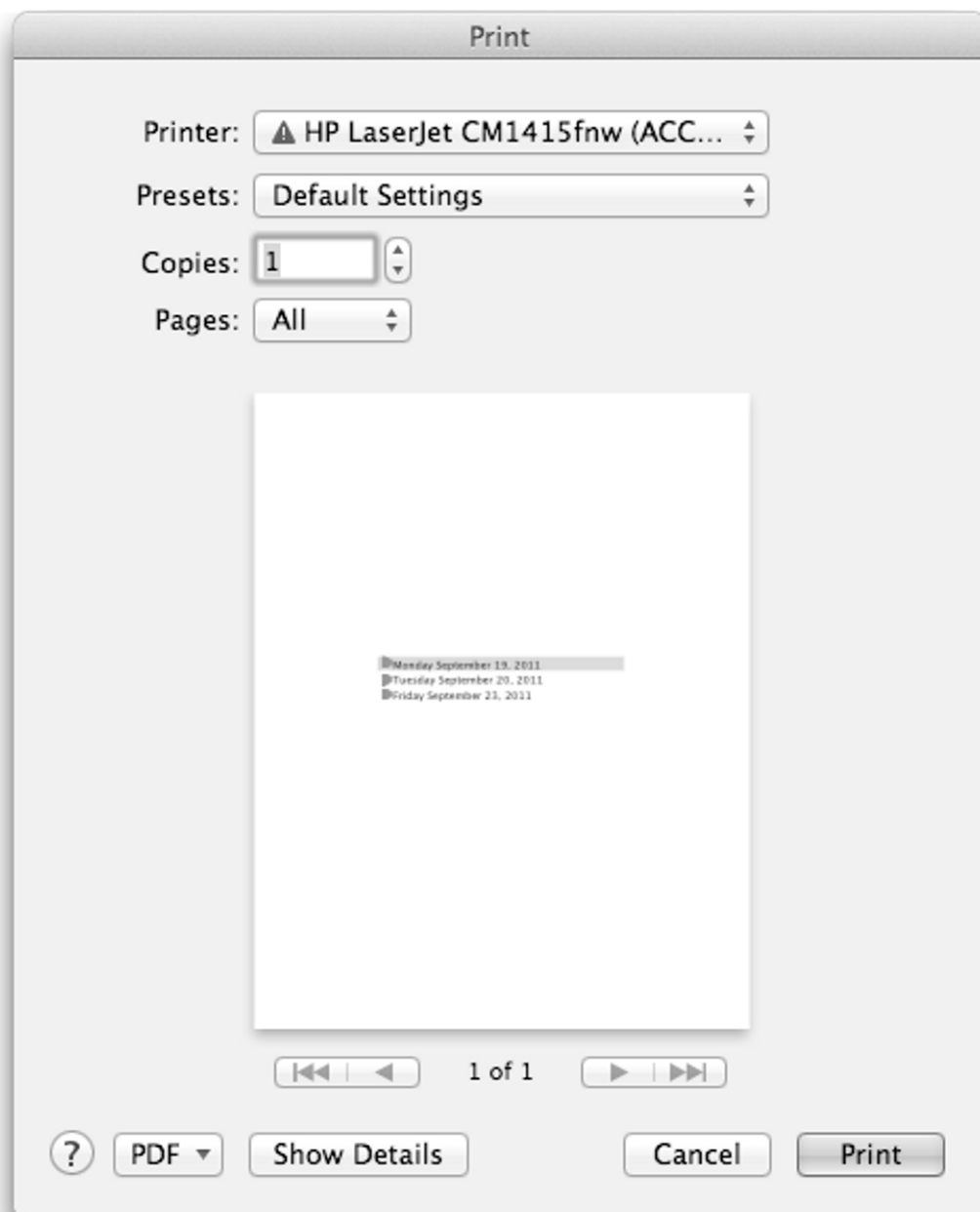


Figure 9–2. The print preview window showing the recently used equations

Graphique should take a clearer approach to printing: it shouldn't depend on where the focus lies for deciding what to print. Further, simply printing the equation that the user typed is probably not what the user would want. Users want to see the graphs of their equations.

Implementing the Method to Print

To print the graph, we print the existing graph view. Open `GraphiqueAppDelegate.h` and declare a `printEquation:` method:

```
- (IBAction)printEquation:(id)sender;
```

Then, open `GraphiqueAppDelegate.m`, import `GraphView.h`, and implement the `printEquation:` method, as shown in Listing 9–1.

Listing 9–1. *printEquation:* in *GraphiqueAppDelegate.m*

```
- (IBAction)printEquation:(id)sender
{
    GraphView *printView = [[GraphView alloc] initWithFrame:NSZeroRect];
    printView.controller = self.graphTableViewController;
    [printView print:sender];
}
```

This method creates a new `GraphView` object and attaches it to the same controller that the `GraphView` used in the user interface uses. By reusing the controller, we can have access to the plot data. Note, however, that the new `GraphView` instance isn't added to any subview. This is because we intend to use it off-screen only. Instead of invoking the `draw:` method, we call its `print:` method to indicate that it needs to paint itself on the printer's graphics context.

Wiring the Method to the Menu

The next step is to rewire the `Print` menu item to the `printEquation:` method. Select `MainMenu.xib` and expand the `Main Menu` to find the `Print` action, as shown in Figure 9–3.

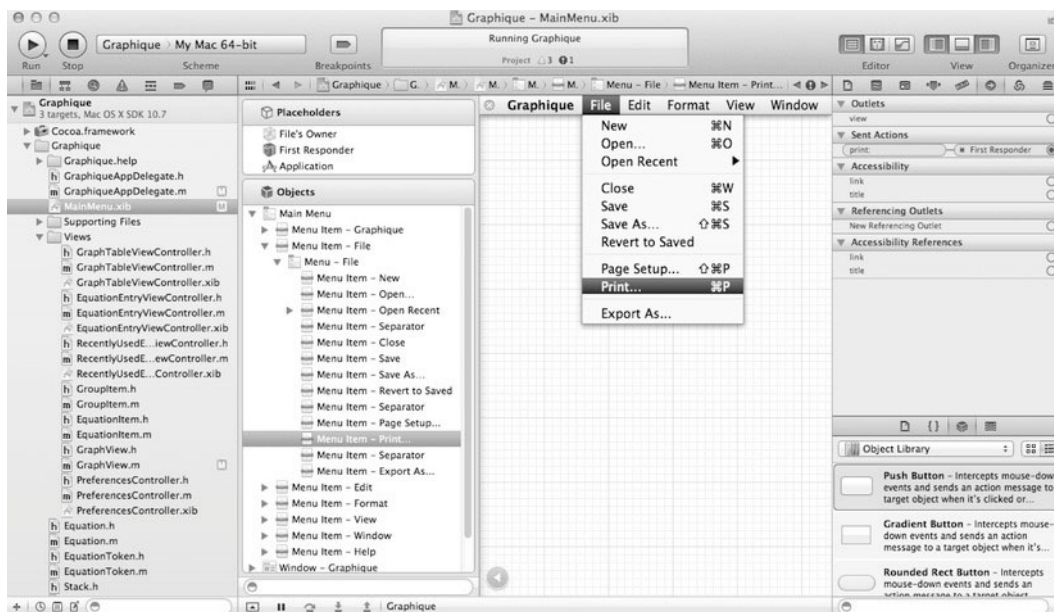


Figure 9–3. Expanding the menu to find the Print item

In the Connections inspector you can see that the action is linked to the `print:` method of the First Responder. This is why when you tried to print before, it printed the equation entry text field (that is, the first responder) when the equation entry field had the focus and printed the recently used equation tree (again the current first responder) when it had the focus. Remove this link by pressing the x next to First Responder in the Connections inspector. It should now look as illustrated in Figure 9–4.



Figure 9–4. *The Print item unlinked*

To attach the print action to the `printEquation:` method, drag from the circle next to `selector` in the Connections inspector to the `Graphique App Delegate` object, and then select `printEquation:`, as shown in Figure 9–5 and Figure 9–6.

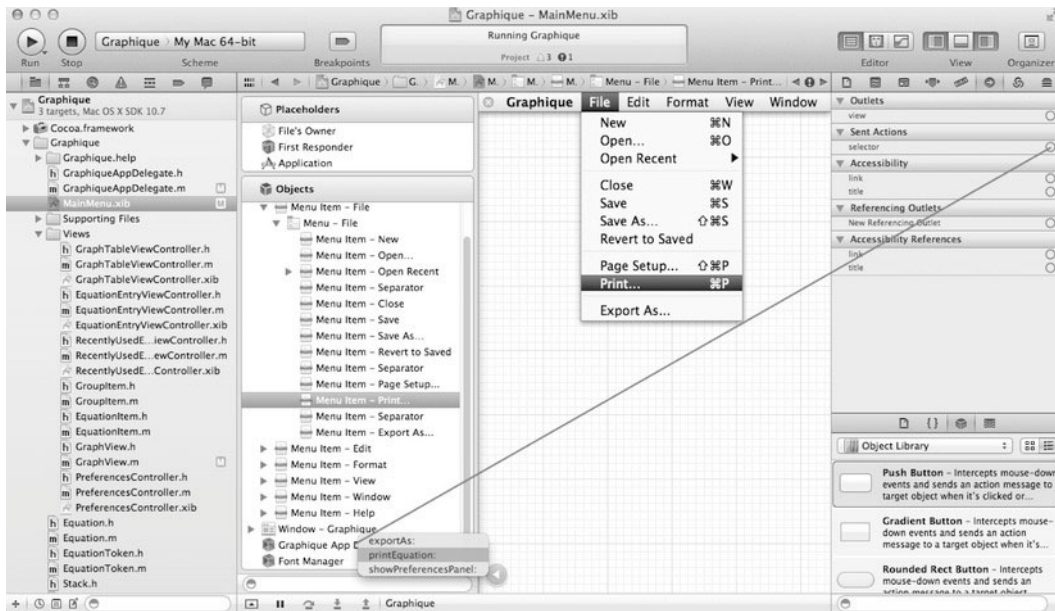


Figure 9-5. Linking the Print item to the `printEquation:` method

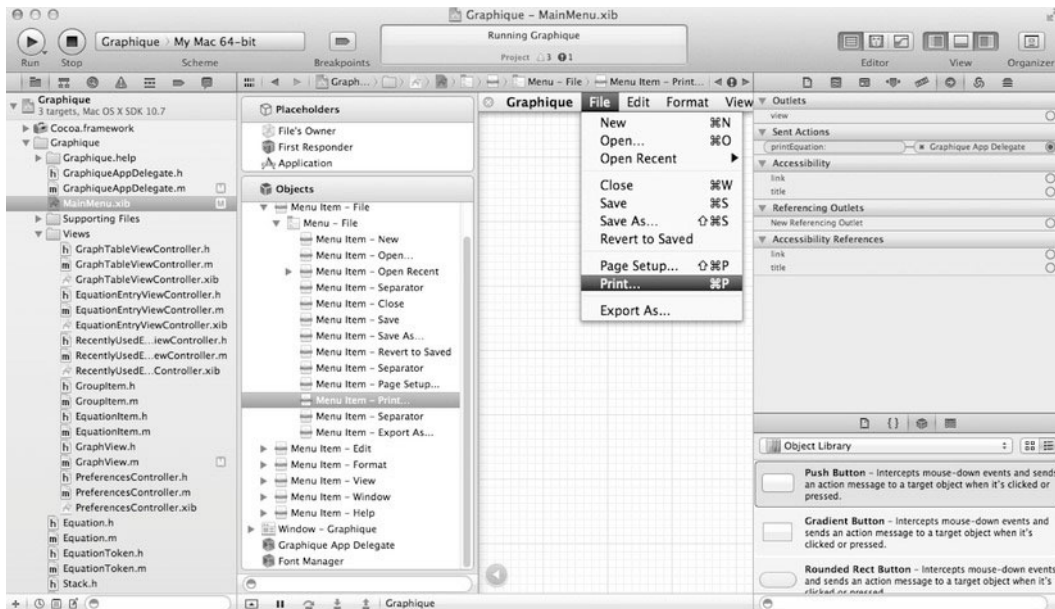


Figure 9-6. The Print item linked to the `printEquation:` method

We also want to enable the Print item on the toolbar. When it is clicked, it sends a `printDocument:` message up the responder chain. To enable the Print item, implement a

`printDocument:` method in `GraphiqueAppDelegate.m` that simply calls the `printEquation:` method, as shown in Listing 9–2.

Listing 9–2. *printDocument:*

```
- (void)printDocument:(id)sender
{
    [self printEquation:sender];
}
```

Sizing the Printed View

If you launch Graphique at this time and select the Print menu item, the print preview shows you that you are about to print a blank page. This is because we sized the offscreen view with `NSZeroRect`, which, like it sounds, is a rectangle with a width and height of zero. At the time of creating the view, we did not know the printer paper settings yet.

Before the view is asked to print itself, users are presented with print options in which they can choose the destination printer and set up the paper layout. The operating system asks our view how many pages it should print by calling `knowsPageRange:` and what their sizes should be by invoking `rectForPage:`. Both of these methods are defined in `NSView` and should be overridden by our custom view.

As users change options, the current print options are put in the current context, and the view receives a `drawRect:` message in order to generate the preview in the print options panel. This allows the print preview to respond to changes in print options.

To add support for printing graphs, open `GraphView.m` and add a method to help the print system know how many pages to print. In our case, we want our graph to fill only one page, so we return a range of `[1; 1]`, as shown in Listing 9–3. At the time when this method is invoked, we know we’re about to print the view, so we size it accordingly.

Listing 9–3. *Returning the Number of Pages to Print*

```
- (BOOL)knowsPageRange:(NSRangePointer)range
{
    NSPrintOperation *op = [NSPrintOperation currentOperation];
    NSPrintInfo *pInfo = op.printInfo;
    NSRect bounds = pInfo.imageablePageBounds;

    [self setFrame:NSMakeRect(0, 0, bounds.size.width, bounds.size.height)];

    range->location = 1;
    range->length = 1;
    return YES;
}
```

We then override `rectForPage:` in order to specify the size of the page (full page). Keep in mind that the `GraphView` instance isn’t the same one that is displayed in the user interface. This one is purely offscreen, and its sole purpose is printing. Listing 9–4 shows the implementation of `rectForPage:` in `GraphView.m`.

Listing 9–4. *Sizing the View for Printing*

```
- (NSRect)rectForPage:(int)page
{
    NSPrintOperation *op = [NSPrintOperation currentOperation];
    NSPrintInfo *pInfo = op.printInfo;
    NSRect bounds = pInfo.imageablePageBounds;

    return NSMakeRect(0, 0, bounds.size.width, bounds.size.height);
}
```

Now start Graphique, graph an equation, and select **File** ► **Print** from the menu. In the print preview window, you see a full-screen graph similar to Figure 9–7.

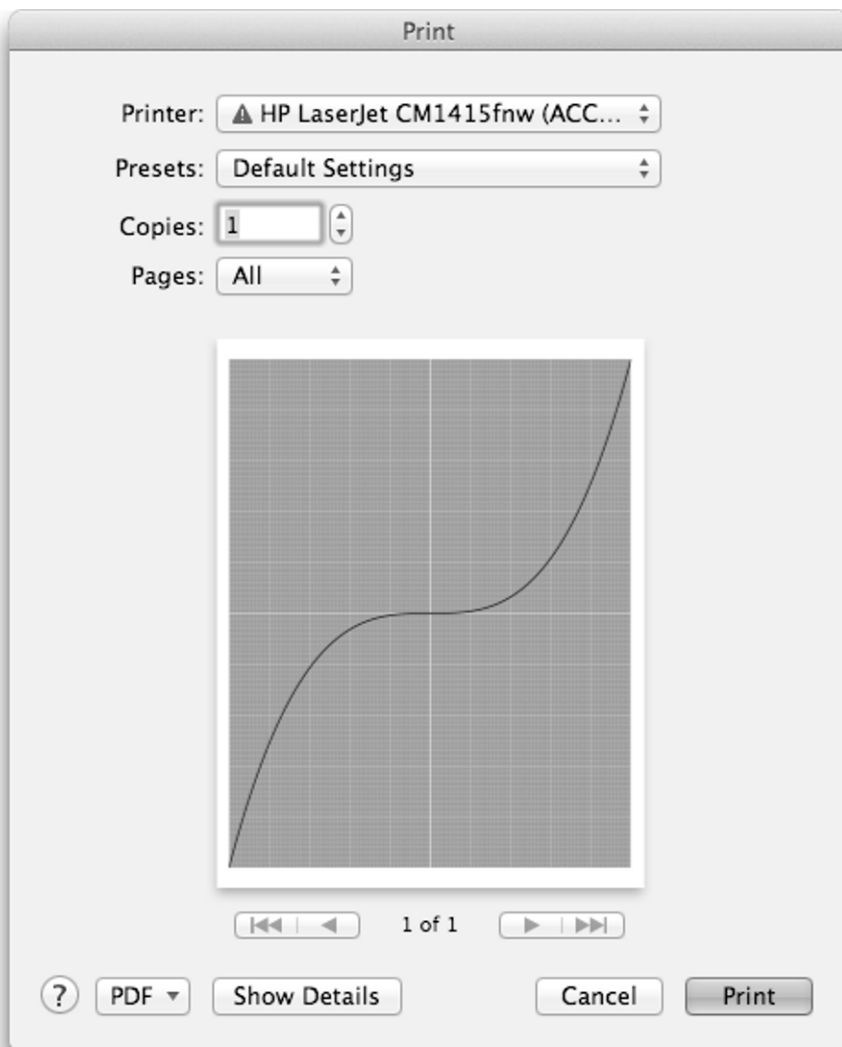


Figure 9–7. *Printing the graph*

Before clicking that Print button, however, read on so you can save yourself some ink.

Drawing for the Printer

Printing the view is nice, but if you want to save some ink, you might want to consider lightening up the print view. For example, we should remove the blue background because it will really suck the ink out of your printer. To remove the blue, customize the view's `drawRect:` method to take different actions based on whether we are drawing to the printer or to the screen.

To determine whether we're printing, look to see whether there is a print operation in the current context:

```
BOOL isPrinting = [NSPrintOperation currentOperation] != nil;
```

We use this flag in the `GraphView`'s `drawRect:` method to adjust the color schemes for printing. Listing 9–5 shows the resulting `drawRect:` method.

Listing 9–5. *The `drawRect:` Method Adjusted for Printing*

```
- (void)drawRect:(NSRect)dirtyRect
{
    ... // code removed to keep book clearer and save some ink

    // Step 2. Paint the background

    // Set the color scheme
    NSColor *background = [NSColor colorWithDeviceRed:0.30 green:0.58 blue:1.0
alpha:1.0];
    NSColor *axisColor = [NSColor colorWithDeviceRed:1.0 green:1.0 blue:0.0 alpha:1.0];
    NSColor *gridColorLight = [NSColor colorWithDeviceRed:1.0 green:1.0 blue:1.0
alpha:0.5];
    NSColor *gridColorLighter = [NSColor colorWithDeviceRed:1.0 green:1.0 blue:1.0
alpha:0.25];

    // Get the line color from the user defaults
    NSData *colorData = [[NSUserDefaults standardUserDefaults] dataForKey:@"lineColor"];
    NSColor *curveColor = (NSColor *)[NSUnarchiver unarchiveObjectWithData:colorData];

    if ([NSPrintOperation currentOperation] != nil)
    {
        // Adjust the color scheme to more greys
        axisColor = [NSColor colorWithDeviceRed:0.2 green:0.2 blue:0.2 alpha:1.0];
        gridColorLight = [NSColor colorWithDeviceRed:0.4 green:0.4 blue:0.4 alpha:1.0];
        gridColorLighter = [NSColor colorWithDeviceRed:0.6 green:0.6 blue:0.6 alpha:1.0];
        curveColor = [NSColor colorWithDeviceRed:0.0 green:0.0 blue:0.0 alpha:1.0];
    }
    else
    {
        // Paint the background
        [background set];
        NSRectFill(dirtyRect);
    }

    // Step 3. Plot the graph
```

```
if(controller.values.count == 0) return;  
... // code removed to keep book clearer and save some ink  
}
```

Figure 9–8 shows the printer-optimized result of drawing the view.

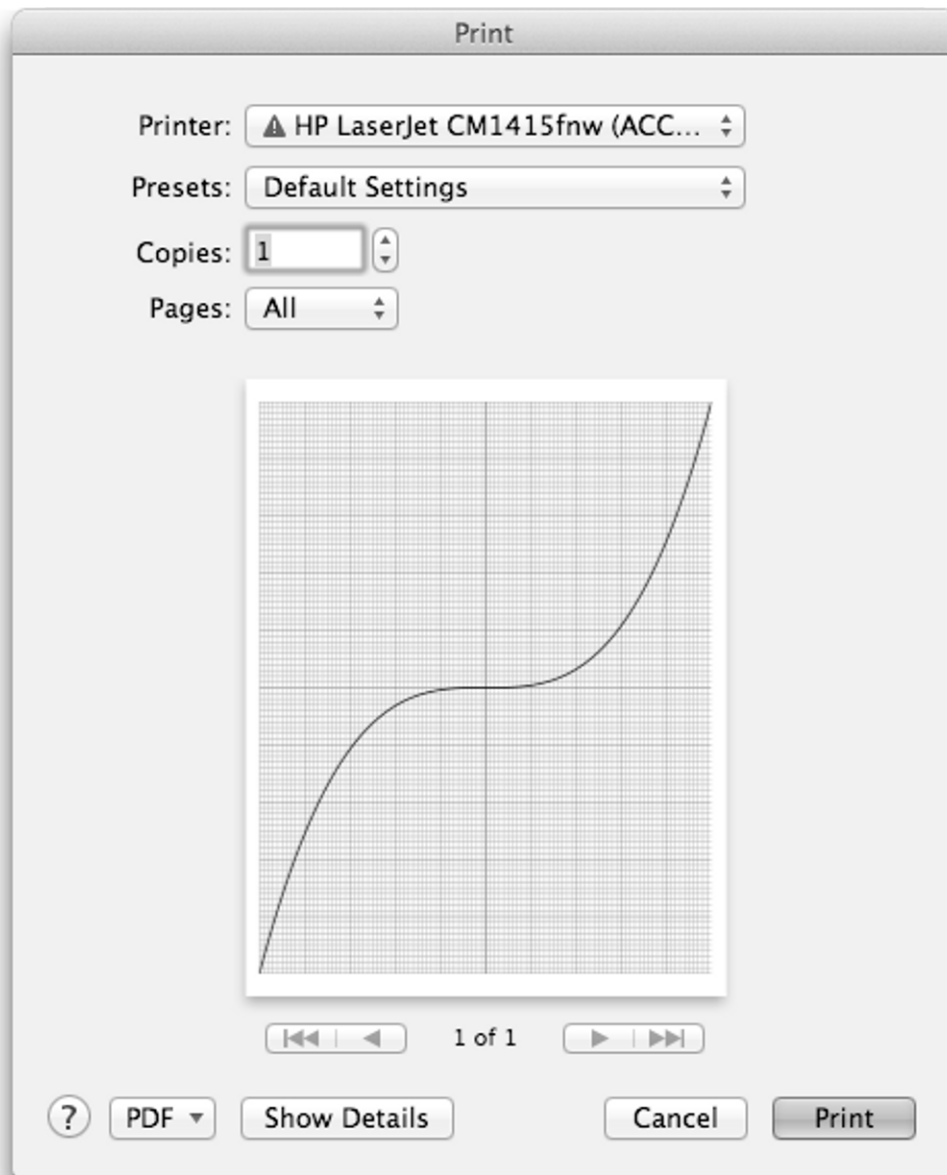


Figure 9–8. *The printer-optimized result of drawing the view*

Feel free to go ahead and print this one. You should get exactly one page, with your graph filling all but the margins.

Spanning to Multiple Pages

Until now, we've used the same view, `GraphView`, for both screen and print views. Often, though, you must use completely different views for printing from those used in the user interface to produce a good printout—especially when the printed version of the data is significantly different from the user interface data. In this section, we want to print the graph view but also the data table that goes along with it. We create a new view to accommodate this need.

So far, we've seen how to print a view to a page, but what happens when the view is so big that multiple pages are required? This is where pagination comes into play. When a view is about to be printed, the print system queries it by calling its `knowsPageRange:` method to find out how many pages ought to be printed. In our previous example, we returned a range of `[1; 1]`. For each page to be printed, the view is asked via the `rectForPage:` method to indicate where the page rectangle is located on the view. In our paginated version, we create a view that is large enough to contain all the pages to print. Each page is a copy of a rectangular area of the view, as shown in Figure 9–9.

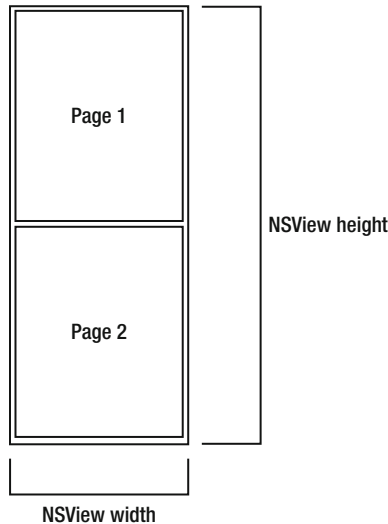


Figure 9–9. *The pages layout on the `PrintView`*

In Xcode, create a new class called `PrintView` that's a subclass of `NSView`. Change the `printEquation:` method in `GraphiqueAppDelegate.m` to utilize the new class, as shown in Listing 9–6. Add an import for `PrintView.h` to the top of `GraphiqueAppDelegate.m`, but leave the import for `GraphView.h` there as well. Note that, until we flesh out the `PrintView` class, `GraphiqueAppDelegate.m` won't compile.

Listing 9-6. *The printEquation: Method Adjusted to Use the Specialized View*

```

- (IBAction) printEquation:(id)sender
{
    PrintView *printView = [[PrintView alloc] initWithFrame:NSZeroRect];
    printView.controller = self.graphTableViewController;
    [printView print:sender];
}

```

Edit `PrintView.h`, as shown in Listing 9-7.

Listing 9-7. *PrintView.h*

```

#import <Cocoa/Cocoa.h>
#import "GraphTableViewController.h"

@interface PrintView : NSView
{
    GraphTableViewController *controller;
    NSFont *font;
    CGFloat heightPerLine;
}

@property (nonatomic, strong) GraphTableViewController *controller;

@end

```

We keep track of the font to use and the height of each line of text in pixels so that we can properly compute the formatting of the pages and the page count.

Calculating the Number of Pages

Then comes the real meat of the print view. Open `PrintView.h`, add a `@synthesize` directive for the controller property, and edit the `initWithFrame:` method to properly initialize the class attributes, as shown in Listing 9-8.

Listing 9-8. *Initializing the PrintView*

```

- (id)initWithFrame:(NSRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        // Set up the font
        font = [NSFont fontWithName:@"Helvetica" size:12.0];

        // The height of a line is the height of the font
        // plus some proportional spacing so the lines
        // are not stuck together.
        heightPerLine = [font capHeight] * 1.5;
    }
    return self;
}

```

To manage the pagination, we have to compute the number of pages by calculating how many lines of data we can fit on a page. We also reserve the first page for the graph. We then resize the view to encompass all the pages. As in the implementation of `GraphView`, the `knowsPageRange:` method is responsible for all this computation. Listing 9–9 shows `PrintView`'s `knowsPageRange:` implementation.

Listing 9–9. *Computing the Number of Pages*

```
- (BOOL)knowsPageRange:(NSRangePointer)range
{
    NSPrintOperation *op = [NSPrintOperation currentOperation];
    NSPrintInfo *pInfo = op.printInfo;
    NSRect bounds = pInfo.imageablePageBounds;

    NSUInteger linesPerPage = bounds.size.height / heightPerLine;
    NSUInteger totalPages = 1 + controller.values.count / linesPerPage;

    totalPages++; // Count a page for the graph itself

    [self setFrame:NSMakeRect(0, 0, bounds.size.width, bounds.size.height * totalPages)];

    range->location = 1;
    range->length = totalPages;
    return YES;
}
```

Determining the Page Size

Now that the number of pages has been computed, we need to be able to specify the rectangular area that each page will cover on the view. We use the schema from Figure 9–9 to lay out the pages. The view's height is big enough to include all the pages so that computing the page area is simpler, as Listing 9–10 demonstrates. We simply move the rectangle down based on the current page being queried. Remember that the page parameter is one-based (in other words, the first page is `page=1`, not zero).

Listing 9–10. *Computing the Page*

```
- (NSRect)rectForPage:(int)page
{
    NSPrintOperation *op = [NSPrintOperation currentOperation];
    NSPrintInfo *pInfo = op.printInfo;
    NSRect bounds = pInfo.imageablePageBounds;

    return NSMakeRect(0, bounds.size.height * (page-1), bounds.size.width,
        bounds.size.height);
}
```

Drawing the Page

Lastly, we implement the `drawRect:` method, which is smart enough to figure out which data to display based on the current page. If the current page is page one, then we simply make a new `GraphView` instance and make it draw itself on the current graphics context. Otherwise, we just paint data on the page. Listing 9–11 shows the entire `drawRect:` method. Be sure to import `GraphView.h` at the top of `PrintView.m` since we are delegating some work to that class.

Listing 9–11. *Printing Each Page*

```
- (void)drawRect:(NSRect)dirtyRect
{
    NSPrintOperation *op = [NSPrintOperation currentOperation];
    NSRect pageBounds = op.printInfo.imageablePageBounds;

    if (op.currentPage == 1)
    {
        // First page, print the graph
        GraphView *graph = [[GraphView alloc] initWithFrame:pageBounds];
        graph.controller = self.controller;
        [graph drawRect: dirtyRect];
    }
    else
    {
        NSUInteger linesPerPage = pageBounds.size.height / heightPerLine;

        // The print operation pages are 1-based. So we remove one to make it
        // zero-based and remove one to make room for the first page (the graph)
        long dataPage = op.currentPage - 2;

        NSDictionary *attributes = [NSDictionary dictionaryWithObject:font
                                                                    forKey:NSFontAttributeName];

        for (int i = 0; i < linesPerPage; i++)
        {
            NSUInteger index = dataPage * linesPerPage + i;
            if (index >= controller.values.count) break;

            NSValue *value = [controller.values objectAtIndex:index];
            NSPoint point = value.pointValue;

            NSRect xRect = NSMakeRect(pageBounds.origin.x, op.currentPage *
pageBounds.size.height - (i+1) * heightPerLine, 100, heightPerLine);
            [[NSString stringWithFormat:@"%%.2f", point.x] drawInRect:xRect
withAttributes:attributes];

            NSRect yRect = NSMakeRect(pageBounds.origin.x + 100, op.currentPage *
pageBounds.size.height - (i+1) * heightPerLine, 100, heightPerLine);
            [[NSString stringWithFormat:@"%%.2f", point.y] drawInRect:yRect
withAttributes:attributes];
        }
    }
}
```

Launch Graphique, enter an equation, and click the Graph button to plot it. Then, select to print it. The print options open up with a preview. You can navigate through the pages of the preview to see how the data is laid out. Figure 9–10 shows the print preview window. We've set it to print four pages per sheet so you can see all the data on one preview window. Go ahead and print this one, as well, to get three pages' worth of your equation as both graph and tabular data.

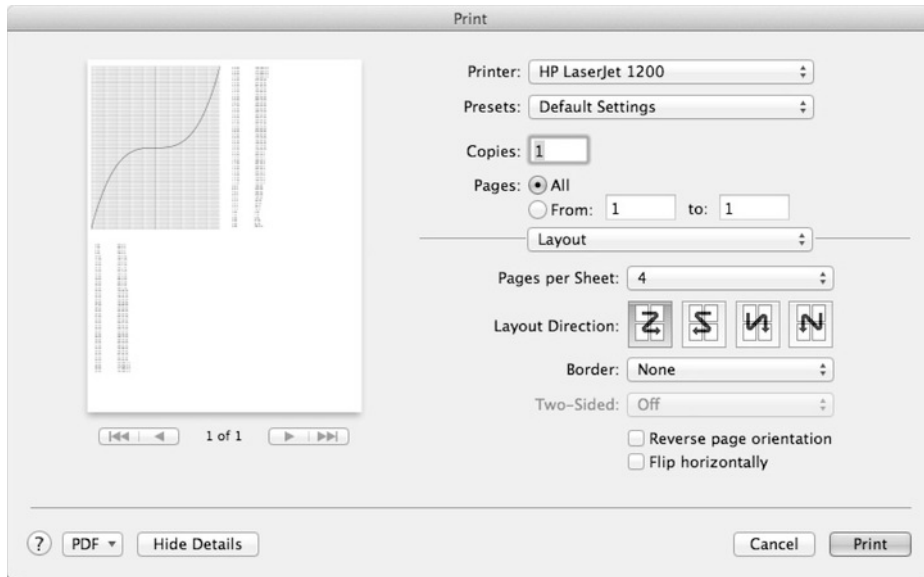


Figure 9–10. *The print preview with pagination*

Summary

Printing can be a difficult task to manage in many languages and platform. Ask any Java programmer, for example, to show you how much work is involved in producing a reasonable-looking printout. Most often, it requires some third-party libraries. In contrast, Mac OS X and Cocoa make this task easy. Printing is second nature to the framework and is supported by default by all views. For some less trivial needs such as pagination, only a little bit more work is required to achieve great results and control how pages are laid out.

This chapter purposely kept the example lean and simple so that you can best adapt it to your needs. Some possible enhancements would be to add a frame to the data table along with column headers. All this would be done with the usual drawing methods, just as if we were drawing them on a regular user interface view.

Submitting to the Mac App Store

With the hard work of development, debugging, and a little more debugging complete, you're ready to apply the final polish to Graphique and submit it to the Mac App Store. This moment is both exhilarating and terrifying. It's exhilarating to be done, to distribute your work, and to perhaps earn some money from your labors. It's terrifying because your work now comes under the scrutiny of millions of Mac users. Will they like it? Will they pan it? How will the reviews turn out? Will they even notice your app amidst the thousands of other apps available on the Mac App Store?

In this chapter, we complete Graphique, applying an icon and sandboxing it, and then we walk you through the Mac App Store submission process. After that, the reviews are up to you and your customers.

Reviewing the Guidelines

Apple maintains a list of guidelines for Mac App Store apps at <https://developer.apple.com/appstore/mac/resources/approval/guidelines.html>. This list can change over time, so be sure to review that list regularly to stay within Apple's guidelines. Apple can and will reject apps that don't adhere faithfully to them.

The list isn't onerously long but is still too long to include here. Despite some of the controversy these guidelines have sparked, nothing on the list should be too surprising. Useful apps that follow the practices in this book, don't do anything out of the ordinary (such as install kernel extensions), don't crash, and don't fake flatulence should sail through the process. Just remember to review the list often.

Finishing the App

Although Graphique is nearly complete, the gap between “nearly” and “absolutely” can cost you sales, and even bar your entry to the Mac App Store. Users don’t want nearly done software; they want software that’s complete. Even though useful software rarely stops growing and changing as you add features, improve workflows, and fix bugs, each release of your software should feel finished.

Terminating Graphique When Its Window Closes

Because Graphique is a single window application, it should terminate if the user closes the window. To add this feature, add the code in Listing 10-1 to `GraphiqueAppDelegate.m`.

Listing 10-1. *Telling Graphique to Terminate If Its Window Closes*

```
- (BOOL)applicationShouldTerminateAfterLastWindowClosed:(NSApplication *)sender
{
    return YES;
}
```

This method will be called when Graphique’s last window, which is its only window, closes. By returning YES, we tell Graphique to terminate.

Adding the Icon

Your application’s icon parades across the Mac App Store, on web sites, in users’ Docks, and in the Mac OS X Finder. The application’s icon offers both a first and a repeated impression on both users and prospective users, and users believe the quality of the icon reflects the quality of the application. Your app’s icon should convey that professionalism and should also indicate the application’s function. Use care when creating or selecting the application’s icon, because a gross misstep could sabotage all your efforts.

If you’re facile enough with graphic design, feel free to create your icon yourself. If not, turn to one of your graphic artist friends, hire a graphic artist, or turn to one of the many crowdsourcing design sites on the Web. However you arrive at your final product, you should have images in the following sizes, measured in pixels:

- 512x512
- 256x256
- 128x128
- 32x32
- 18x18 (not in the `.icns` file—used for the status item)
- 16x16

Generally, the icons should depict the same image but at different resolutions and likely different levels of detail (especially for the smaller icons). The Graphique icon, for example, uses the same image at different sizes, except for the smallest icons; the 18x18 icon and the 16x16 icon ditch the grid, x, and y labels.

Apple publishes guidelines for your icons at <http://developer.apple.com/library/mac/#documentation/UserExperience/Conceptual/AppleHIGuidelines/IconsImages/IconsImages.html> that demonstrate Apple's legendary attention to detail. This document provides invaluable information; we recommend you read it periodically to make sure your icons fit the Mac ethos.

Once you have your images assembled, you use the Icon Composer application to combine them into a single icon file with an .icns extension. Launch Icon Composer, found in /Developer/Application/Utilities. You should see a screen with some empty squares, ready to accept your image files, as shown in Figure 10-1.

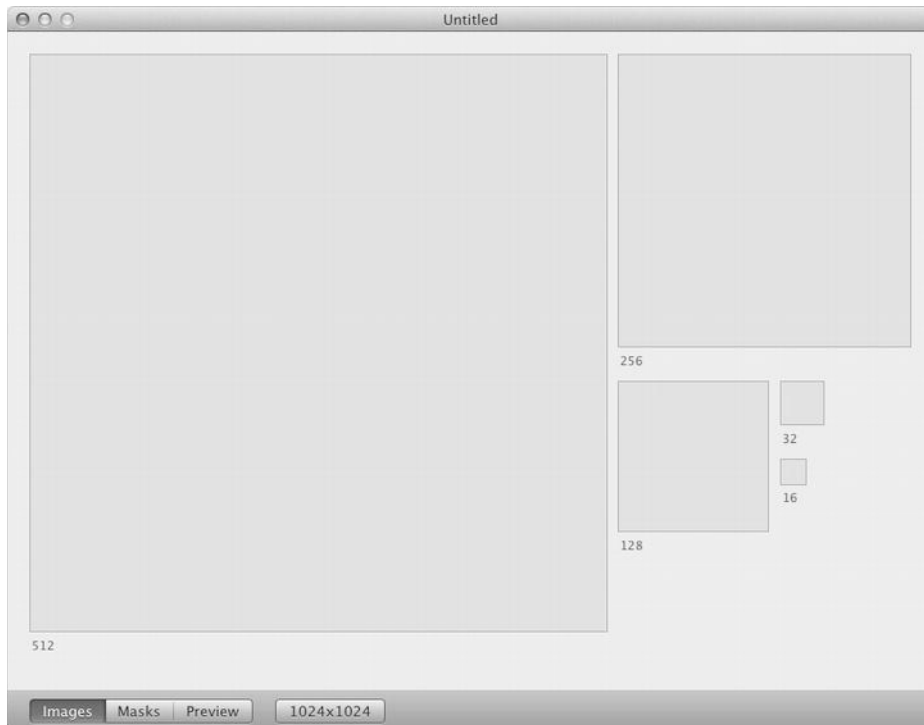


Figure 10-1. *Icon Composer with a blank window*

Drag each of your images to the appropriate rectangles, one at a time, until all the rectangles are filled, as shown in Figure 10-2. Notice that the images for 512, 256, 128, and 32 are all resizings of the same image, while the image for 16 has the details reduced and is the image we used for bookmarks for the Graphique help book, built in Chapter 8.

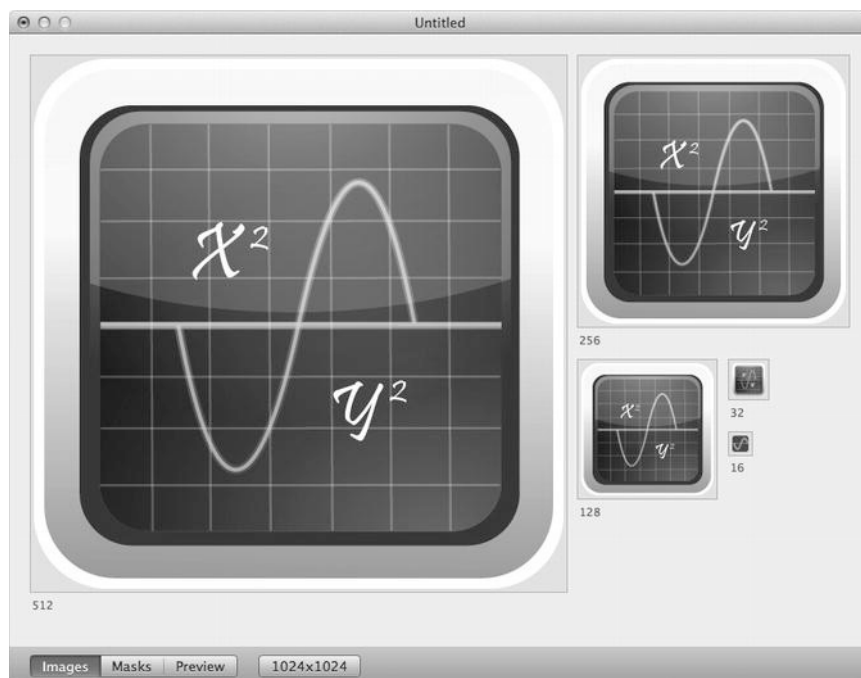


Figure 10-2. *Icon Composer with images added*

Save this file as `Graphique.icns` somewhere on your local drive. Then, in Xcode, select your project in the Project navigator, select the Graphique target under Targets, and select the Summary tab. You should see an empty image well labeled App Icon, as shown in Figure 10-3. Drag and drop `Graphique.icns` from the Finder to the App Icon image well to copy the icon to the project and set it for the application, as shown in Figure 10-4.

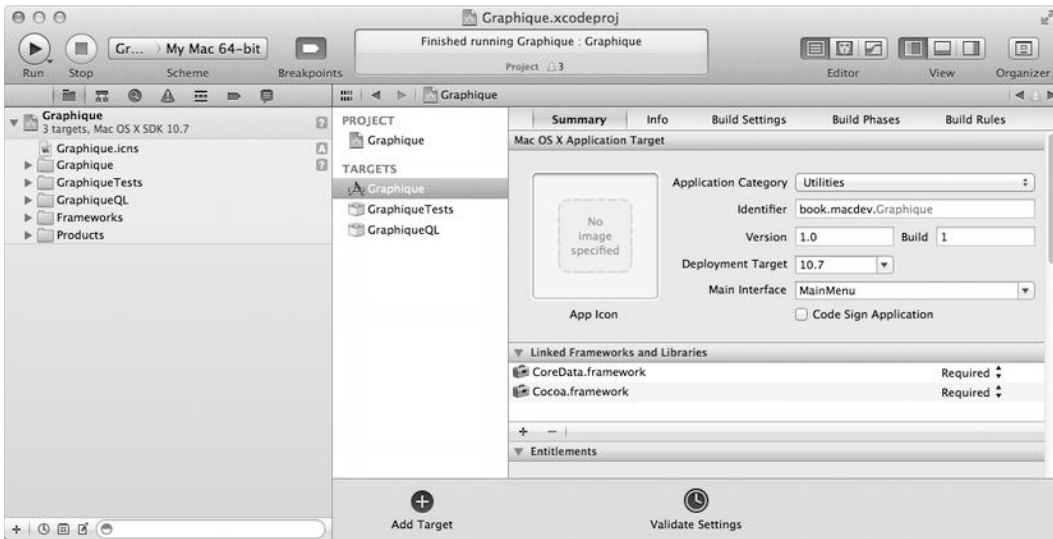


Figure 10-3. Preparing to set the App Icon

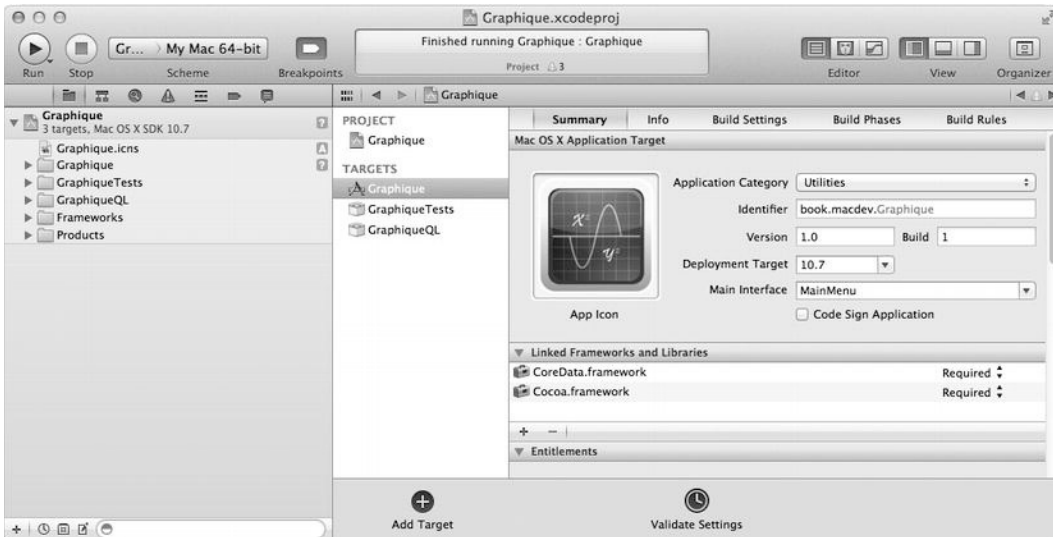


Figure 10-4. Graphique with the App Icon set

Reviewing the Property List File

With the project and target still selected, click the Info tab to display Graphique's property list file. This file is mostly correct at this point, but the copyright string it contains is the one Xcode generated at the outset of the project, which may or may not be correct. Check the text of the copyright string, in the key "Copyright (human-readable)," and update as appropriate. Do the same for the GraphiqueQL target.

Cleaning Up the Menu

When we first generated the code for Graphique, Xcode generated a menu for us. As we've grown Graphique, we've made some changes to that menu. Some unused or incomplete vestiges remain, however, so in this section we square the menu from what it is to what it should be.

Finishing the About Box

If you run Graphique and select **Graphique ► About Graphique** from the menu, you see its About Box, as shown in Figure 10-5. You can see the effects of the changes you've made in this chapter: it displays the icon and copyright you just configured.

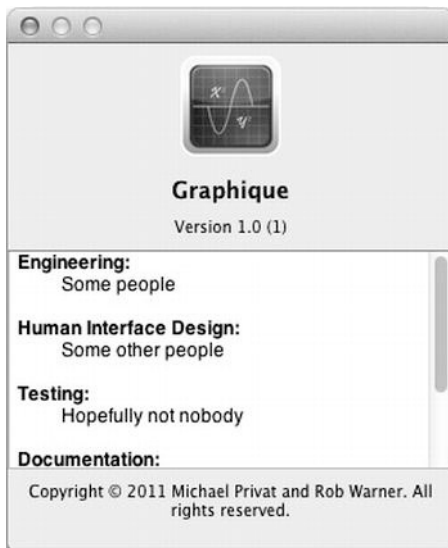


Figure 10-5. *Graphique's About box*

Although the top and bottom parts of the About Box show correct information, the middle remains clearly unfinished. Graphique pulls that text from a file called `Credits.rtf`, found in the group Supporting Files. Open that file in Xcode and modify it appropriately. You can preserve the Xcode-generated headings and put appropriate information in those sections, or you can change the file entirely. See Figure 10-6 for an example.

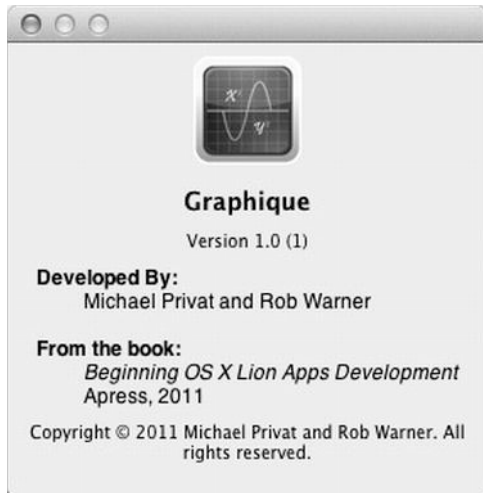


Figure 10-6. *The updated About box*

If the text you enter fits in the allotted space, its background will be gray, and no scroll bar will be present. Longer text, however, will have a white background with a scroll bar.

Centering the Preferences Panel

When users select **Graphique** ➤ **Preferences** from the menu, the preferences panel displays in the lower left of the screen, which is kind of odd. Let's change it to display in the center of the screen. Open `PreferencesController.xib` in Xcode, select **Panel – Preferences** under **Objects**, and select the **Size** inspector. In the **Initial Position** section, select **Center Horizontally** and **Center Vertically** in the two drop-downs, as shown in Figure 10-7.

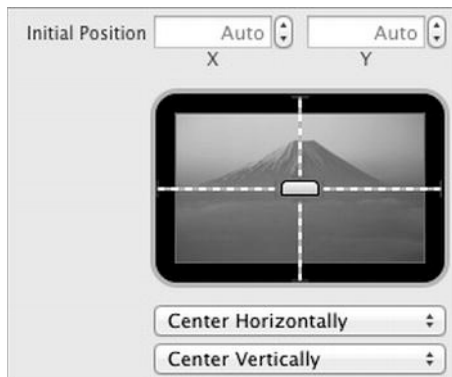


Figure 10-7. *Centering the preferences panel*

Build and run Graphique, and then select to display the preferences panel to confirm that it appears centered on the screen.

Removing Unused Menu Items

Not every menu item generated by Xcode has meaning for Graphique, and so several remain unimplemented. Rather than leave them in the menu, as if they did something, remove them. To remove them, select `MainMenu.xib` in Xcode, select the item menu to remove, and press the Delete key on your keyboard. The items to remove are as follows:

- File ► New
- File ► Open Recent
- File ► Close
- File ► Page Setup...
- Edit ► Undo
- Edit ► Redo
- The separator that was below File ► Redo
- Format ► Text

Cleaning up the menu to include only implemented items is always a good practice. If future versions of your application add features that require those menu items, you can always add them back.

Adding a Menu Item for Full-Screen

Graphique supports OS X Lion's full-screen mode, offering users the standard full-screen button in the upper-right corner of its window. To conform with Apple's interface guidelines, however, it should also offer a menu item and key equivalent for entering full-screen mode. This item should be called Enter Full Screen and have the keyboard equivalent `Ctrl+⌘+F`. The Enter Full Screen menu item should be under the View menu (note: applications that don't have a View menu should put this item under the Window menu).

To create the Enter Full Screen menu item, expand the View menu in Interface Builder, and drag a Menu Item instance from the Object Library beneath it. In the Attributes inspector, type **Enter Full Screen** for Title and press the keys `Ctrl+⌘+F` for Key Equivalent. See Figure 10-8 for guidance.

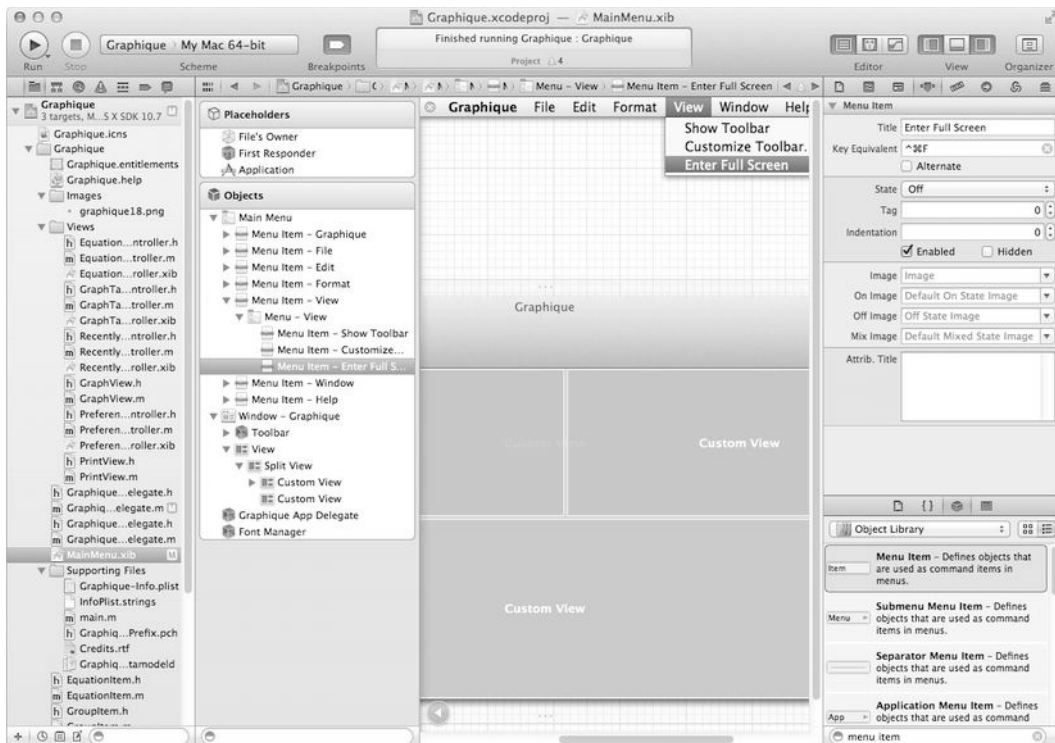


Figure 10-8. Adding the Enter Full Screen menu item

The action that the Enter Full Screen menu item should trigger is `toggleFullScreen:`. To connect the menu item to the action, Ctrl+drag from the menu item to the First Responder item and select `toggleFullScreen:` from the menu. If you build and run Graphique, you can enter full-screen mode by selecting **View** ► **Enter Full Screen** or by pressing `Ctrl+⌘+F`. To exit full-screen mode, press `Esc` or `Ctrl+⌘+F`.

Setting the Initial Window Size and Location

When users first launch Graphique, the initial size of the window is too narrow to adequately display the equation field, as shown in Figure 10-9. Sure, they can resize the window, but the equation entry field is the most important part of the window, at least until a graph displays, so we should make it more visible.

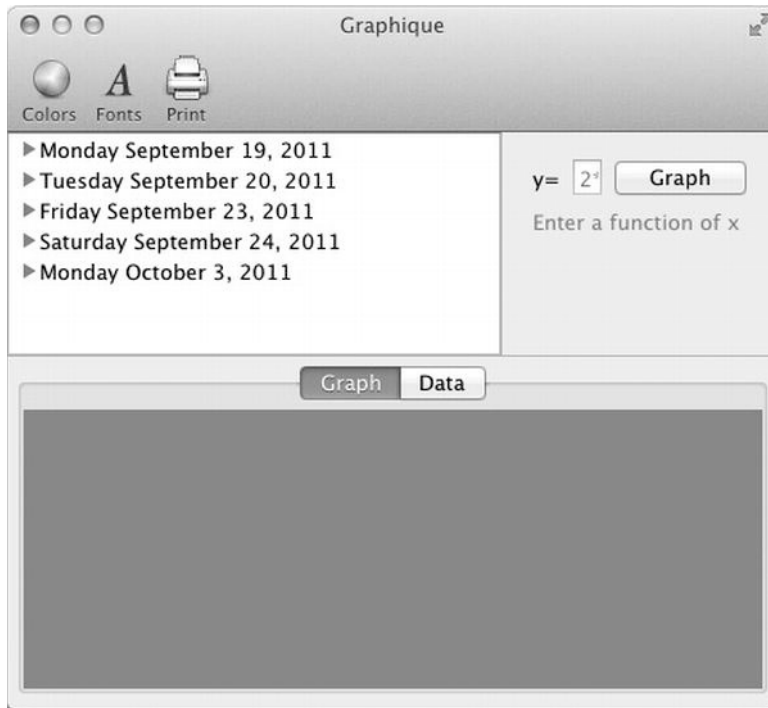


Figure 10-9. *The initial Graphique window*

To set the initial size and position of the Graphique window, open `MainMenu.xib` in Xcode, select `Window - Graphique`, and open the Size inspector. The Initial Position option, set to Proportional Horizontal and Proportional Vertical, is fine. Depending on where the window shows on the screen, though, you can drag it to a better location—perhaps toward the upper-left corner. In the Size section, though, change the width field from 480 to 800, as shown in Figure 10-10. Build and run Graphique to see your changes, as shown in Figure 10-11.

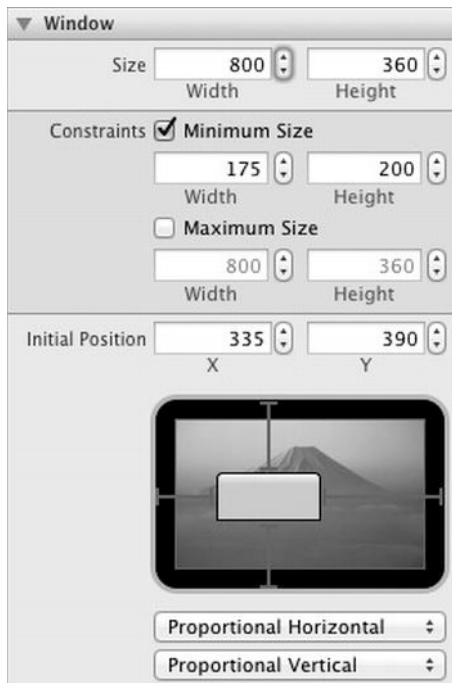


Figure 10-10. Setting the width to 800

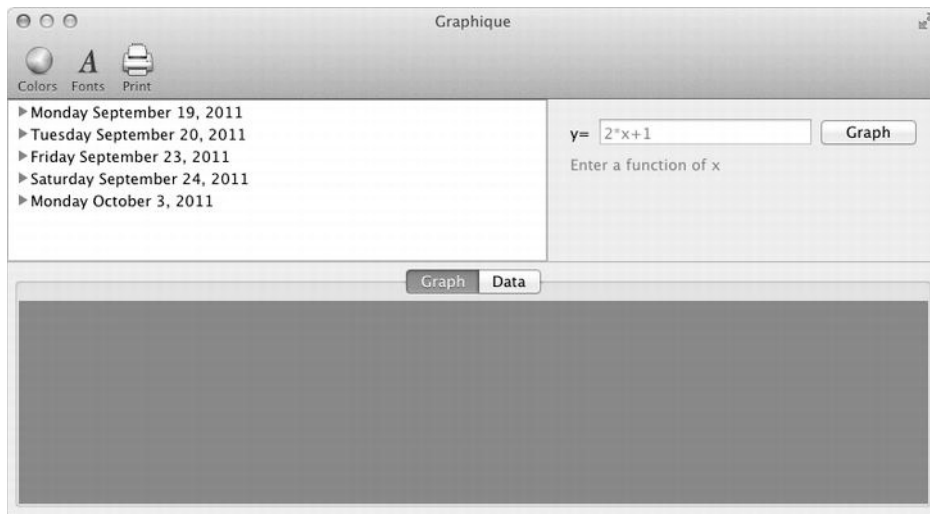


Figure 10-11. The Graphique window set to 800 pixels wide

However you initially size and position the window, however, you should respect that your users might have their own ideas for sizing and positioning. To embrace their preferences, use OS X's Autosave feature to remember how the user sizes and positions Graphique's window so you can restore it on launch. To use this feature, select

MainMenu.xib in Xcode, select the Window – Graphique item, and open the Attributes inspector. In the Autosave field, enter **MainWindow**. Graphique will now store the window's size and position in its user defaults and will automatically restore the window to that size and position when Graphique launches.

Signing the Code

When you grab a pen and scrawl your signature on a check, your mortgage document, or an 8x10 glossy of yourself, you attest approval of the intent of the thing you signed. Signing code plays a similar role, allowing you to digitally attest that you authored the code. The operating system can detect if digitally signed code has been altered after its creation, whether by hackers, viruses, or file corruption. This provides users an additional layer of safety when using your application.

Signed code establishes that it hasn't been altered, that it comes from a specific source, and that it can be trusted for a specific purpose, such as accessing keychain data from a previous version of the same application. Submissions to the Mac App Store must all be digitally signed.

Using the Developer Certificate Utility

To begin the code-signing process, go to <http://developer.apple.com> and sign in to your account. If you have not yet joined the Mac Developer Program, you can do so at <http://developer.apple.com/programs/mac/>. As of this writing, it costs \$99 per year and is required to distribute your apps through the Mac App Store.

Once logged in, go to the Developer Certificate Utility page at <http://developer.apple.com/certificates/index.action>, as shown in Figure 10-12. From this page, you create App IDs for the apps you develop, create and download certificates for establishing your identity, register systems for ad hoc distributions of your apps, and create provisioning profiles to marry App IDs, certificates, and optionally systems together to distribute your applications.

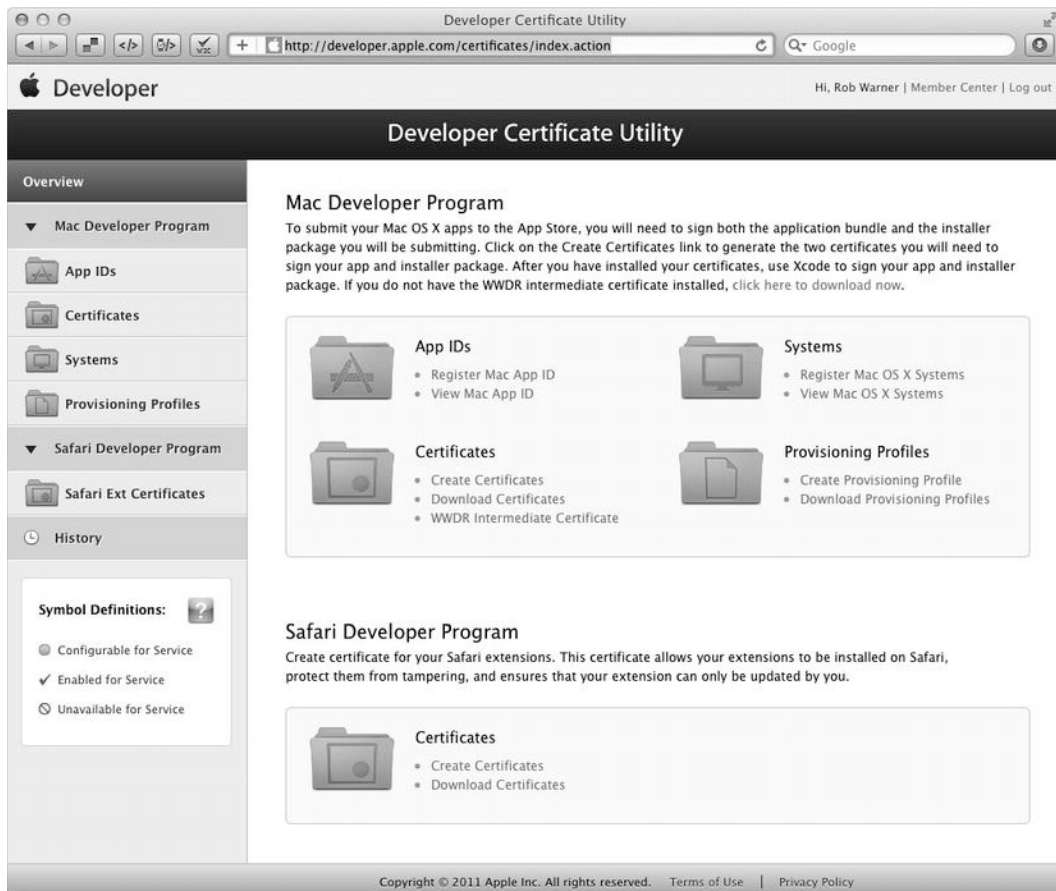


Figure 10-12. The Developer Certificate Utility screen

Registering Your App IDs

Each app you create carries a globally unique App ID that matches the bundle identifier specified in your app's property list file. Remember that this is usually your domain name, in reverse order, followed by the name of your app in lowercase.

Click the Register Mac App ID link to see the form for registering your App ID. The first field, Name or Description, is for your benefit, to identify this App ID in a friendly way so you remember which app you've tied the App ID to. For Graphique, we've entered Graphique for the Name or Description field, and book.macdev.Graphique in the Bundle Identifier field, as shown in Figure 10-13. If you're following along, note that you can't register that same App ID because we already did.



Figure 10-13. Registering Graphique's App ID

When you click Continue, you see a confirmation screen that allows you to review the App ID you're registering. Review it carefully, because you can't edit a registered App ID; you can only view it and delete it. If satisfied, click Submit. Apple will register your App ID, which you can review at any time by clicking the App IDs link on the left of the Developer Certificate Utility screen, as shown in Figure 10-14.

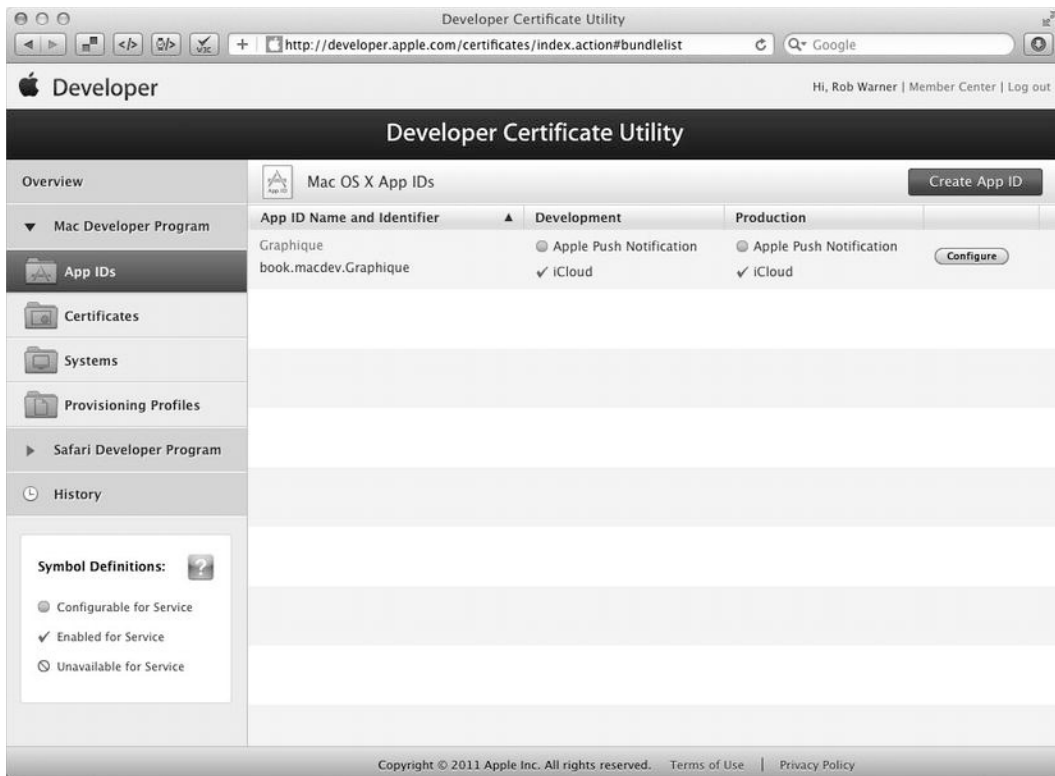


Figure 10-14. The registered Graphique App ID

Since Graphique doesn't support iCloud services, you must verify that the App ID isn't configured for iCloud. To do that, click the **Configure** button, and if the box titled **Enable for iCloud** is checked, uncheck it and click **Done**.

Installing Apple's Certificate

Apple provides a certificate it calls the World Wide Developer Relations (WWDR) certificate that ties the certificate you will create back to Apple. Apple is the root certificate authority that vouches for your certificate. To install this certificate, return to the main Developer Certificate Utility screen by clicking the **Overview** link on the left. Underneath the **Certificates** section you see a link that says **WWDR Intermediate Certificate**. Click the link to download the certificate in a file called `AppleWWDRCA.cer`, and then double-click it in **Finder** to install it. Open the **Keychain Access** app on your computer, found in the **Utilities** folder under the **Applications** folder, and look in the login certificates to find the certificate called **Apple Worldwide Developer Relations Certification Authority** you just installed.

Creating Certificates

You can click the Overview link on the left to return to the main Developer Certificate Utility screen shown in Figure 10-11 to reveal the Create Certificates link, or you can click the Certificates link in the left menu to reveal the Create Certificate button. However you get there, click to create a certificate. You'll see the screen shown in Figure 10-15.

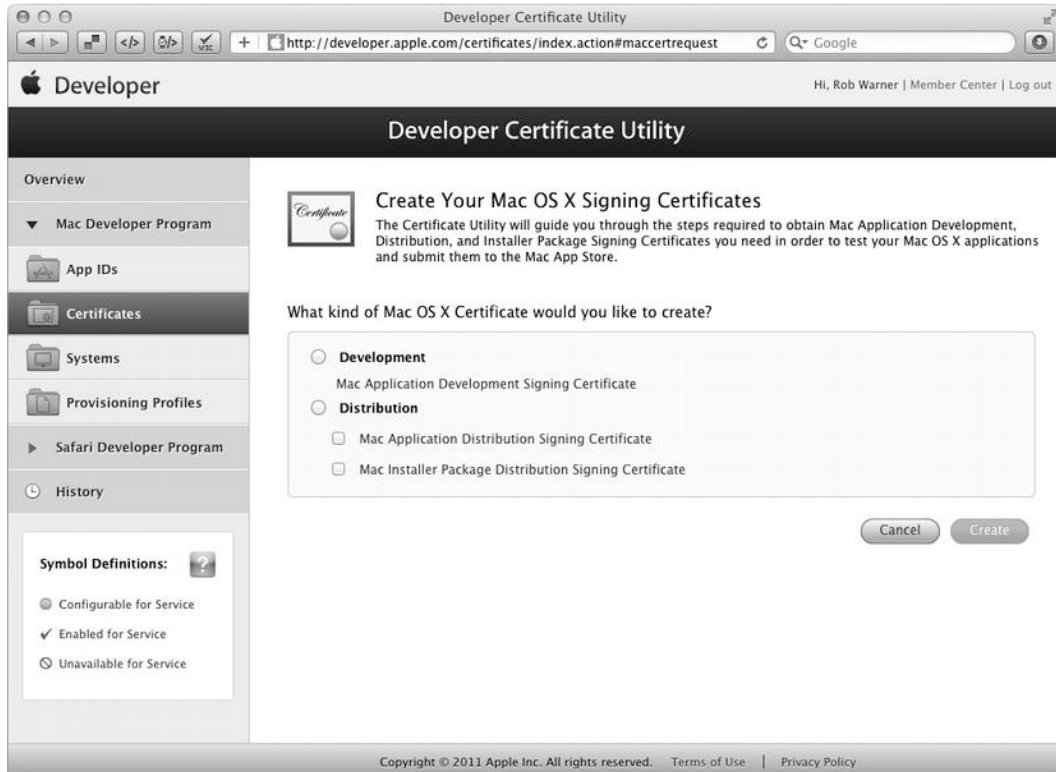


Figure 10-15. *Creating certificates*

You can either create a development certificate, useful for distributing code-signed apps to a limited audience for testing, or create distribution certificates. Each Mac App Store app requires two certificates: one for the application and one for its installer. Since we're creating certificates to submit to the Mac App Store, select the radio button next to Distribution, make sure both check boxes are checked, and click Create.

The next screen, shown in Figure 10-16, instructs you open the Keychain Access app on your computer to generate a certificate-signing request. Follow the steps it gives you:

1. Within the Keychain Access drop-down menu, select **Keychain Access** ► **Certificate Assistant** ► **Request a Certificate from a Certificate Authority**.
2. In the Certificate Information window, enter the following information:

- In the User Email Address field, enter your e-mail address.
 - In the Common Name field, create a name for your private key (e.g., John Doe Dev Key).
 - In the Request is group, select the “Saved to disk” option.
3. Click Continue within Keychain Access to complete the CSR-generating process.

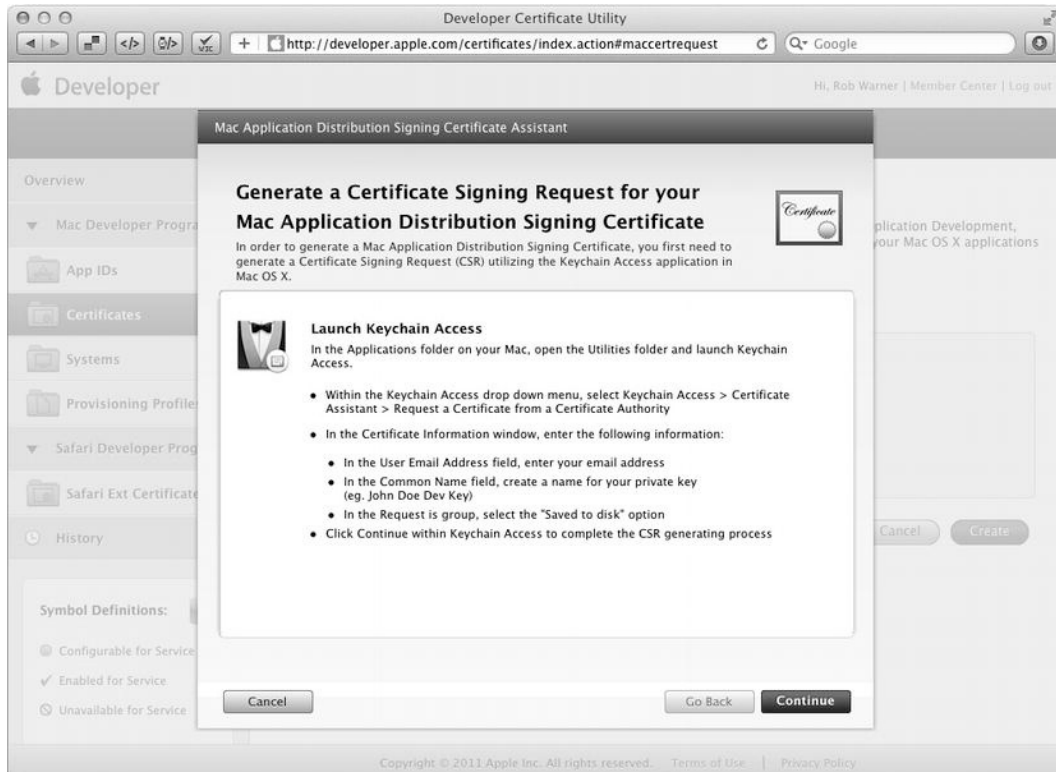


Figure 10-16. Generating a certificate-signing request

When you select the menu item **Keychain Access** ► **Certificate Assistant** ► **Request a Certificate from a Certificate Authority** from the Keychain Access app, you see the screen shown in Figure 10-17.



Figure 10-17. *Entering certificate information for requesting a certificate*

Enter your information, making sure to select the “Saved to disk” option, and click Continue. It will prompt you to save the file, as shown in Figure 10-18. Click Save to save it to your desktop, and you’ll see the dialog shown in Figure 10-19 affirming that your request was saved. Click Done to dismiss it.



Figure 10-18. Saving your certificate-signing request



Figure 10-19. Confirmation that your request was saved

Return to the Developer Certificate Utility in your web browser, and click Continue. You'll see the screen shown in Figure 10-20 requesting you to upload the certificate-signing request file you just created. Click the Choose File button, select the file you created, and then click the Generate button to upload your request and generate the certificate. After a few moments, you'll see the screen shown in Figure 10-21 telling you that your certificate was generated.



Figure 10-20. *Uploading the certificate signing request file*

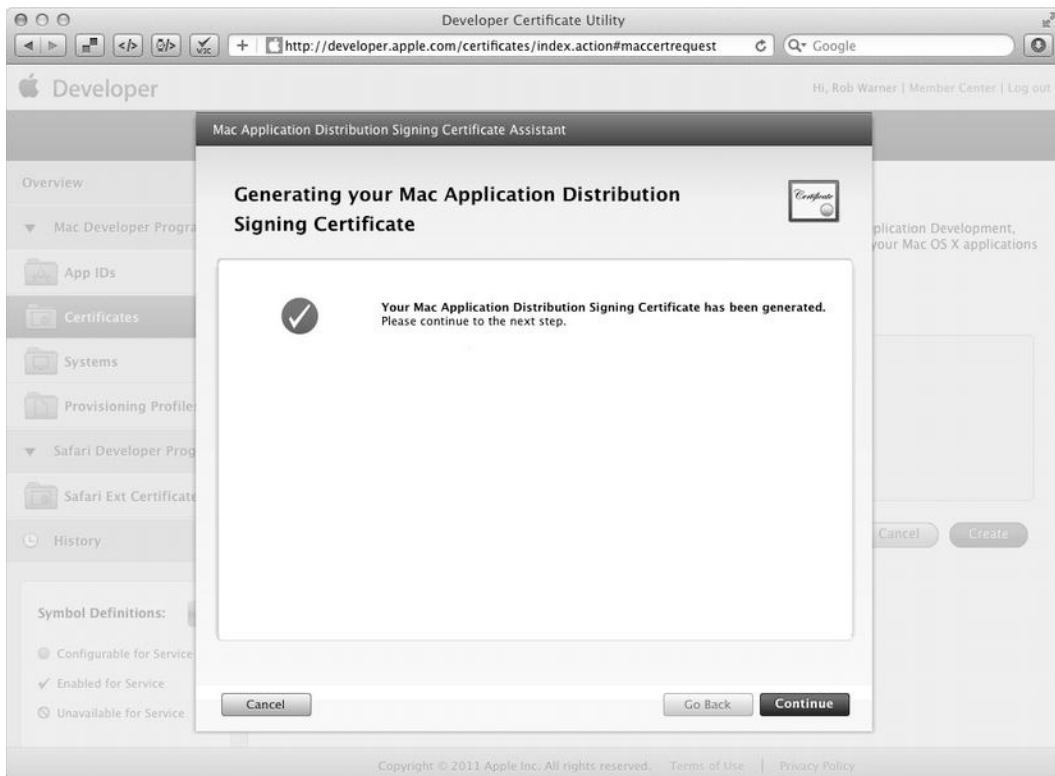


Figure 10-21. Seeing that your certificate was generated

Click Continue to see the screen shown in Figure 10-22, which allows you to click the Download button to download your certificate. Download your certificate and then double-click the certificate file in Finder to install the certificate. Then, click Continue.



Figure 10-22. *The certificate download screen*

The Developer Certificate Utility takes you through the same steps again, only this time for your Mac Installer Package certificate. Go through the same steps again to generate the certificate signing request, upload it, download the certificate, and install the certificate. When you've completed these steps, verify that both certificates were installed by opening the Keychain Access app, selecting login under Keychains and My Certificates under Category. You should see your newly installed certificates, as shown in Figure 10-23.



Figure 10-23. *The certificates installed*

Creating the Provisioning Profile

A provisioning profile ties an App ID and certificates together. You can also tie the profile to specific systems by first creating the systems using the Systems section in the Developer Certificate Utility. This is useful for creating profiles to distribute your applications on an ad hoc basis to certain users, usually for testing purposes. Since

we're uploading Graphique to the Mac App Store, though, and want it available to all systems, we don't include specific systems in the provisioning profile.

Click the Provisioning Profiles link on the left to see any existing profiles you have, as shown in Figure 10-24. Click the Create Profile link on that screen to begin creating your profile.

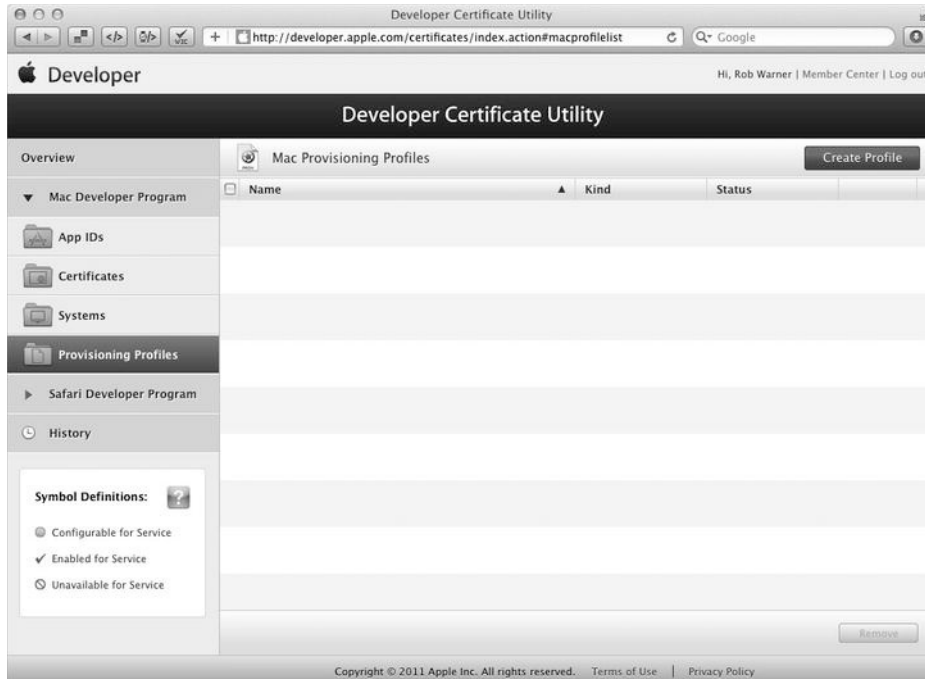


Figure 10-24. The Provisioning Profiles screen

The next screen, shown in Figure 10-25, asks for some information for the provisioning profile. For Kind, select Production Provisioning Profile. This adds a section to the screen for which certificate to use, as shown in Figure 10-26. Since we have only one production certificate, that one is listed. If we had selected Development Provisioning Profile, our development certificates would have been listed.

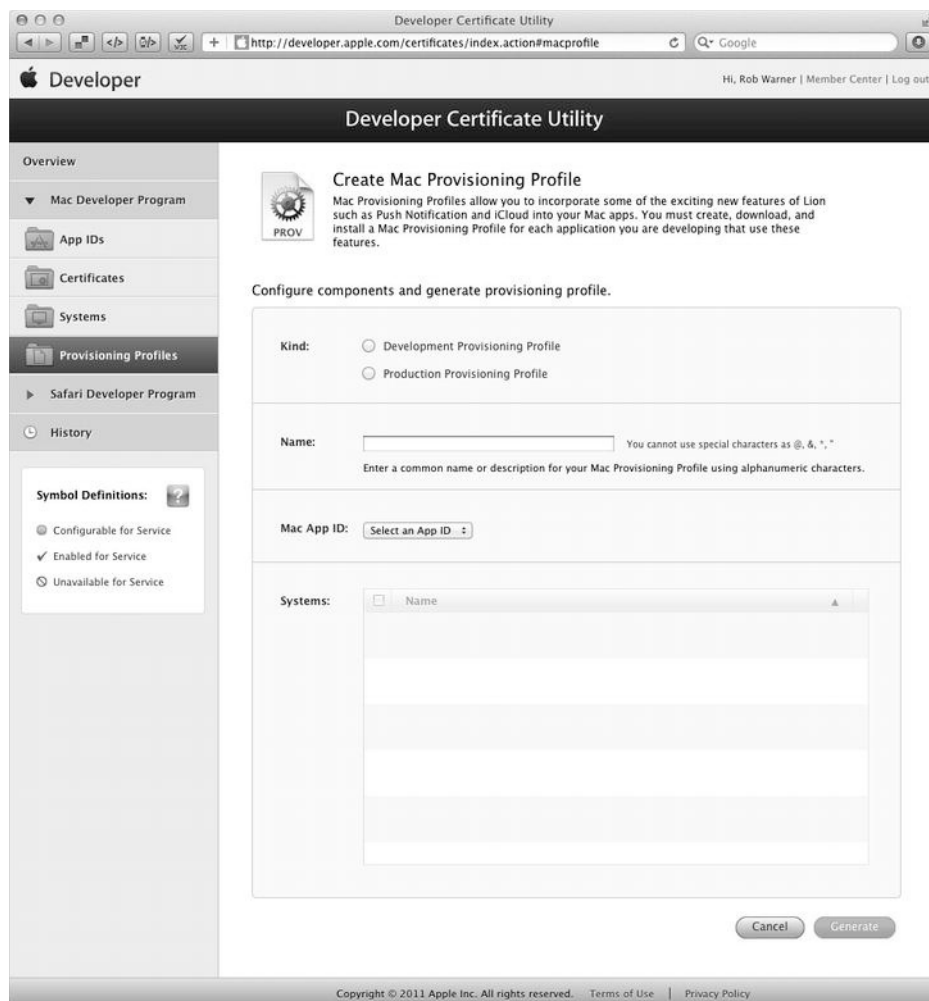


Figure 10-25. *Creating a provisioning profile*

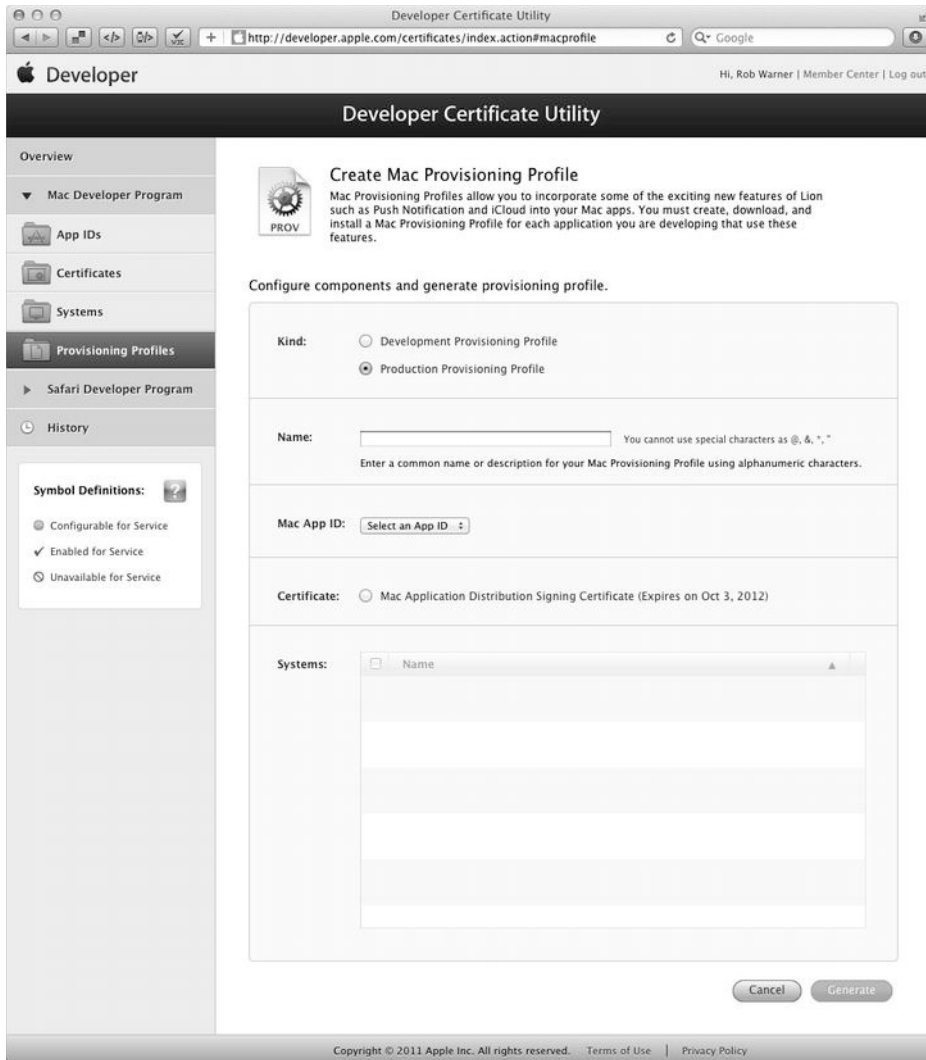


Figure 10-26. Creating a production provisioning profile

Continue to fill out the screen. For Name, enter a friendly name for the profile. For Graphique, we entered **Graphique Production Distribution**. Select the App ID for your app, select the proper certificate, and click Generate. Your provisioning profile is generated, and then you're prompted to download and install the provisioning profile. Click the Download button. The provisioning profile downloads, and then the Profiles preference pane automatically launches and asks you if you want to install the provisioning profile, as shown in Figure 10-27. Click Cancel, because you can't install this profile on your system: this profile doesn't have any system IDs in it. It's the profile for the production distribution, remember? You must install it in Xcode, however, so that you can use it to sign your build. You can do this by dragging the file onto the Xcode Organizer window, onto the Provisioning Profiles section, or clicking the Import button

on the Xcode Organizer window and following the instructions to import the provisioning profile. You can also drag the provisioning profile file from the Finder and dropping it on the Xcode icon in the Dock.



Figure 10-27. Being prompted to install the provisioning profile

Configuring the Build to Sign the Code

Finally, you must configure the build to sign the code using the provisioning profile you just created. Go to Xcode and select your project in the Project navigator and the Graphique target under Targets. Select the Summary tab, and in the Mac OS X Application Target section, you should see a check box that says Code Sign Application. Check it, as shown in Figure 10-28. This will automatically select the provisioning profile and configure your builds to use it.

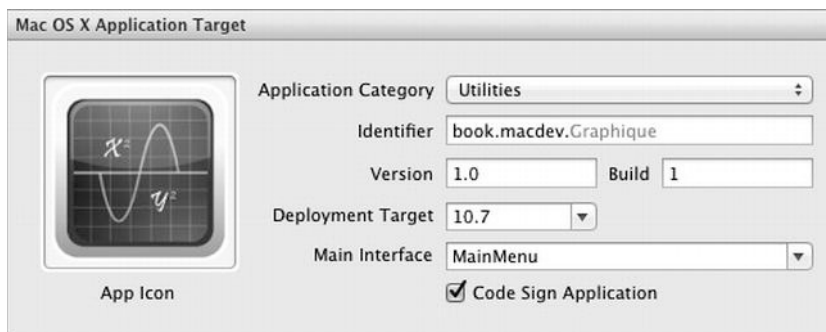


Figure 10-28. Configuring the build to code sign the application

For most of your applications, checking this box is all that's required for configuring code signing. Because of the Graphique QuickLook plug-in that Graphique contains, however, you must do a little extra work to get the proper code-signing configuration. With the Graphique target still selected, click the Build Settings tab and find the Code Signing section. For the Other Code Signing Flags, enter the following:

```
-i book.macdev.Graphique
```

This tells Xcode to ignore specific App IDs and use Graphique's App ID when code signing all bundles within the Graphique target, which includes the Graphique QuickLook plug-in.

You must also configure the QuickLook plug-in to sign the code as part of its build. Xcode doesn't provide a nice Code Sign Application check box for a QuickLook plug-in, so instead you must configure the code signing in the build settings. Select the GraphiqueQL target and the Build Settings tab. Find the section called Code Signing and, for the Code Signing Identity field, select 3rd Party Mac Application under Automatic Profile Selector (Recommended). The Code Signing section should match Figure 10-29.

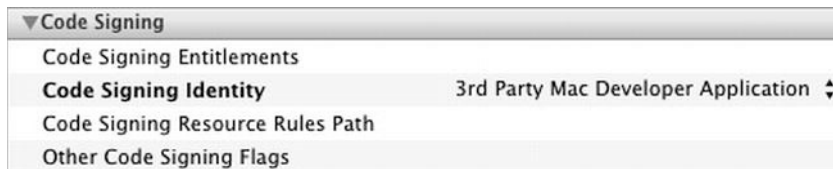


Figure 10-29. Signing the Graphique QuickLook plug-in code

You must do one more thing for the Graphique QuickLook plug-in. With the GraphiqueQL target still selected, find the section called Deployment and the entry that says Skip Install and set it to Yes.

NOTE: Don't forget to set Skip Install to Yes for the Graphique QuickLook plug-in.

Sandboxing the App

Apple has introduced sandboxing to OS X application development as an additional security measure to curb applications from malicious behavior and is requiring that all submissions to the Mac App Store, as of November 2011, implement sandboxing. This section explains sandboxing and its companion concept, *entitlements*, and sets up Graphique's sandbox and entitlements.

What Is Sandboxing?

By default, OS X applications run with all the rights and privileges of the user account under which they run. This means that anything you can do on your Mac without entering your system password, any running app can do: silently, clandestinely, without

any notice or plea for permission. Can you delete songs in your iTunes library? So can an app. Can you send an e-mail to your boss? So can an app. Can you, in a moment of delirium, zip up your documents containing private information like bank account numbers and medical history and post them on a web site for identity thieves to peruse? So can an app, and it doesn't even require the delirium.

We rely on the honor of developers not to do any of these things, and the vast majority don't—at least not intentionally. Whether wrongdoing happens by intent or coding error, however, the effects of bad application behavior can range from annoying to devastating. Sandboxing reins in bad behavior by limiting what an application can do. It provides the app a metaphorical sandbox, isolated from the rest of the operating system, and prevents it from the following activities:

- Reading and writing files
- Accessing the network, whether incoming or outgoing
- Accessing connected hardware devices:
 - Camera
 - USB drives
 - Printers
 - Microphones
- Reading or writing user data from the address book, calendars, or location information

Many applications can perform in such an environment, although the list of restrictions is onerous and is designed for security, not utility. Many more applications, however, can't abide all the restrictions that a sandbox imposes. Imagine, for example, a word processor that can't save, open, or print documents, or a web browser that can't open web sites.

To escape the restrictions of sandboxes in a controlled way, Apple provides what it calls entitlements.

What Are Entitlements?

Entitlements grant specific permissions to applications that sandboxing prevents. As a developer, you build in the entitlements your applications receive. Sandboxing takes all privileges away, and entitlements give specific privileges back. Using entitlements, you can restore an application's ability to read and/or write files (even to specific folders, like the Download or Music folder), access the network (incoming, outgoing, or both), print, see the world through the camera, and a handful of other privileges. See Apple's documentation on Entitlement Keys at <http://developer.apple.com/library/mac/#documentation/Miscellaneous/Reference/EntitlementKeyReference/EnablingAppSandbox/EnablingAppSandbox.html> for more specific information on what privileges you can grant your application.

By sandboxing an application from the rest of the computing environment and then entitling it to access specific computing resources, you establish a trust between developer and user that establishes what the application can and cannot do.

As we set up Graphique's sandbox, we must grant it privileges to do the following:

- Read and write files
- Print documents

Graphique should have no other entitlements.

Establishing the Sandbox

To establish a sandbox, you create what's called an entitlements file that contains information about what your application is entitled to do. The entitlements file is a property list file for which Xcode provides a point-and-click interface. Select the project in the Project navigator, the Graphique target under Targets, and the Summary tab. Expand the Entitlements section, which looks like Figure 10-30.

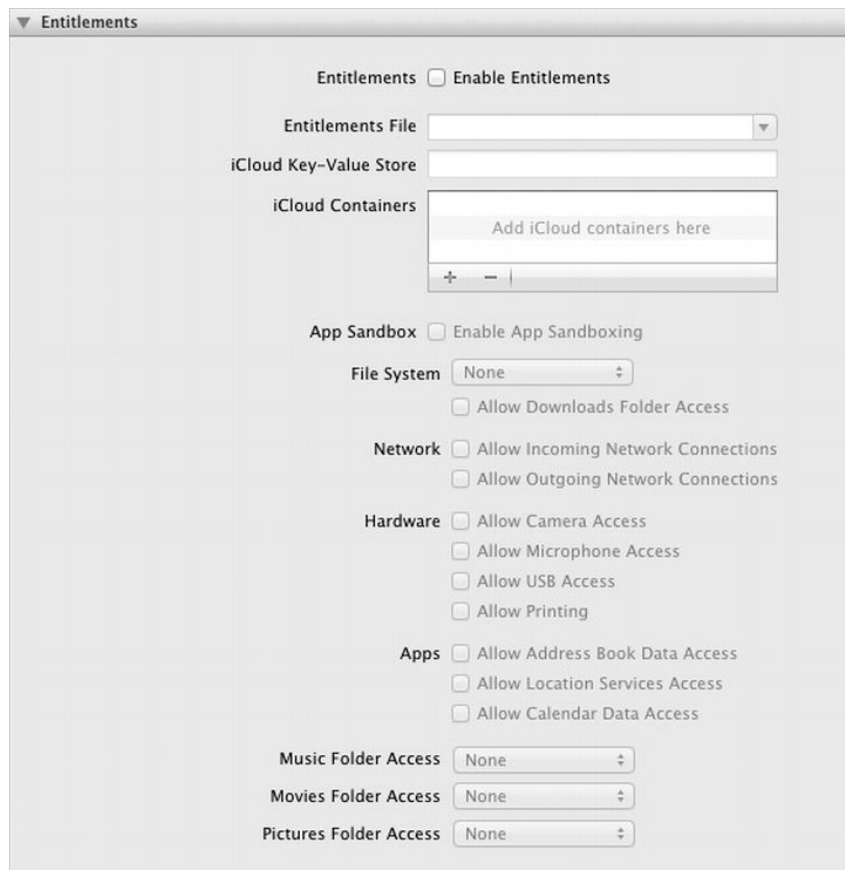


Figure 10-30. The Entitlements settings before configuration

Check the Enable Entitlements check box, and Xcode fills out a few values for you. Leave the Entitlements File set to Graphique, but erase the iCloud Key-Value Store entry. Select book.macdev.Graphique in the iCloud Containers section and click the minus button to delete it. Change the File System setting to Read/Write Access and check the Allow Printing check box. Your settings should match Figure 10-31.

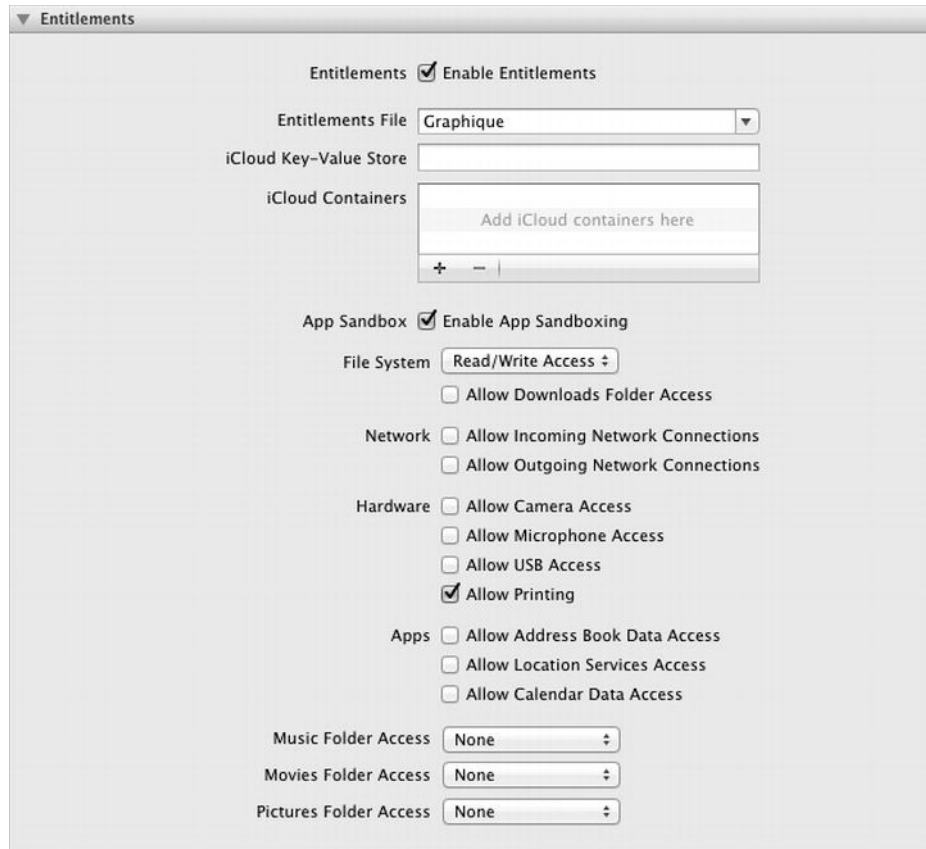


Figure 10-31. *The configured entitlements for Graphique*

You'll notice that Xcode added a file called `Graphique.entitlements` to the project that contains this configuration information as a property list.

Building for Release

With everything configured, you're ready to build the app for release. Select **Product** ➤ **Archive** from the Xcode menu to build and archive the release. Your computer will churn a bit and then display Xcode's Organizer window with the Archives tab selected, as shown in Figure 10-32.



Figure 10-32. Archive for Graphique

We'll come back to this archive later when we're ready to submit Graphique to the Mac App Store.

Setting Up Your Web Site

Each app in the Mac App Store provides a link to the app's web site so that users can learn more about the app and the company or developers behind it and get support for the app. The web site can be as simple or as complex as you want, using whatever technologies you want. This is your web site running on your servers; Apple just links to it from the Mac App Store.

Apple does provides some artwork for you to use on your web site and guidelines for the usage of that artwork, though. The artwork is available at <http://developer.apple.com/appstore/mac/resources/marketing/>, and the guidelines are found at <http://developer.apple.com/appstore/mac/MacAppStoreMarketingGuidelines.pdf>. You should read and understand those guidelines before setting up your web site. Read those guidelines carefully to make sure you understand and follow them.

Using the Artwork

Apple provides an “Available on the Mac App Store” logo, in Photoshop format, for you to use on your web site. The image helps you follow the spacing guidelines set forward by Apple by providing the appropriate margin around the logo. Open the image, resize it to fit your web site (minimum: 40 pixels high for the visible logo, not including the margin), hide the Background and Notes layers, and export it as a PNG (or whatever format you want to use).

Apple also provides images of computers so you can display screenshots of your application as if it were running on a computer. The images come in two versions: one for showing screenshots as windows and one for showing screenshots in full-screen mode. You can use the full-screen ones only for apps that are designed to run in full-screen mode.

Whichever set of images you use, follow the instructions in the “Placing Your Application Screen on the Apple Product Image” section in the guidelines document referenced earlier.

Creating the Web Site

Your app’s web site is an opportunity to sell your app, so focus on what your app does and why users should go download it from the Mac App Store. This isn’t a marketing book, so that’s all we’ll say about that.

You should link the “Available on the Mac App Store” logo to your app in the Mac App Store. To get the URL, after your app is configured in iTunes Connect (which you’ll do later in this chapter), you can get a link to your app on its summary screen—it’s the link behind View in App Store, under Links, as shown in Figure 10-33.

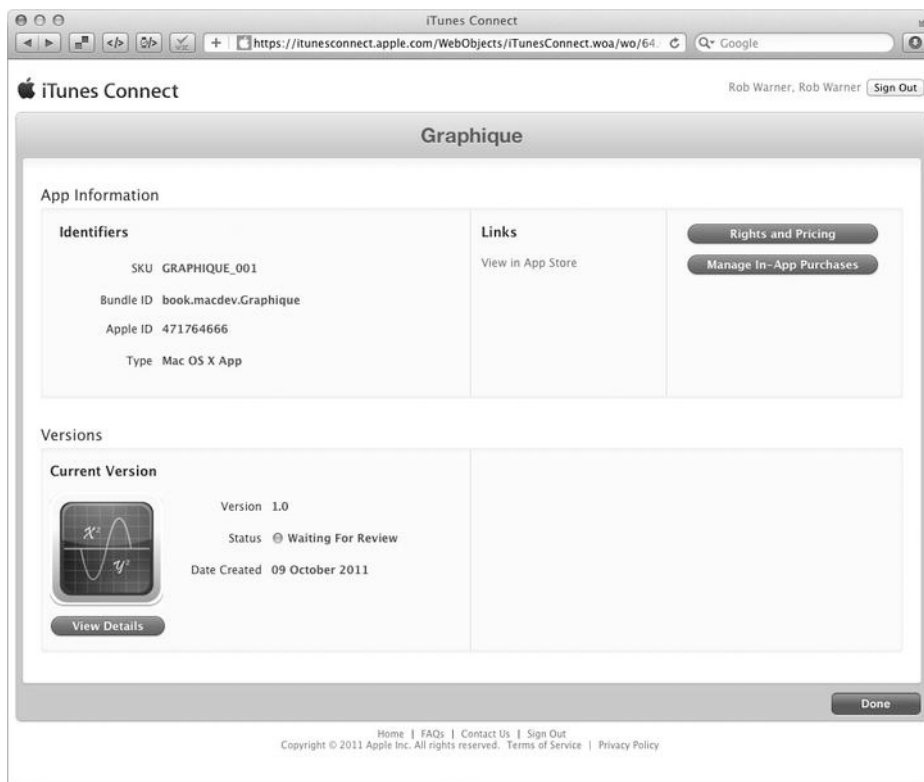


Figure 10-33. *Getting the link for the Mac App Store*

Figure 10-34 shows Graphique's web site, available at <http://grailbox.com/graphique>.



Figure 10-34. Graphique's web site

Submitting the App

You still have a fair amount of work to do to submit your app, so plan your schedule for more than just a few minutes to get this done. The next sections walk you through the app submission process.

Setting Up Your iTunes Connect Account

Your iTunes Connect account allows you to manage the apps you have in the Mac App Store. In it, you specify information about your apps, such as the Mac App Store category to display them in, and you specify details that allow you to get paid, such as your bank account information. You can then use your iTunes Connect account to get download and sales information, among other things, for your apps.

To set up your iTunes Connect Account, point your web browser to <http://itunesconnect.apple.com/>, and enter your Apple ID and password when prompted. Follow the instructions to set up a contract for Mac OS X Paid Applications, including contact information and associated bank account information.

Uploading Your Application

After all the work of developing and testing your application, you're finally ready to upload it. Click the Manage Your Applications link to see a list of all your existing apps in both the Mac App Store and the iOS App Store. To upload a new app, click the Add New App button and select Mac OS X App to begin creating your new app.

Setting the App Name

If this is your first app in an Apple App Store, you'll next be asked for your primary language and the developer name to display for all your apps on Apple's App Stores. Since you can't change these values once set, choose these items with care.

On the next screen, enter an app name and an SKU number, and select the Bundle ID of your app (which you should have already configured using the Developer Certificate Utility). For App Name, enter the name of your app (Graphique, in our case). For SKU Number, enter some string that will be unique to your app. Large companies have more need for this field than small ones, and this can be whatever you want. We enter **GRAPHIQUE_001**, imagining that if we ever release a Pro version of Graphique we can use SKU **GRAPHIQUE_002** for that. For Bundle ID, select your previously configured Bundle ID. You can also click the link to register a new Bundle ID, which will throw you back to the Developer Certificate Utility.

Selecting Availability and Pricing

The next screen has you select your availability and pricing for your app. The availability defaults to the current date, which you should leave unless you have some reason to launch your app on a specific date. For pricing, click the link to view the pricing matrix to understand the pricing tiers. The tier you select establishes the price for the app in the many countries iTunes does business in. For Graphique, we select Tier 1, which means it costs 99 cents in the United States and 85 yen in Japan, for example. Click Continue to proceed to the next screen.

Entering Additional Information

The next screen requests loads of information, so you'll be happier if you've prepared some of this before you get here. Some of this information should match information in your app's property list. Fill out the fields appropriate to your app. Read the discussion of each of the sections to understand what the fields mean and how we filled them out for Graphique.

Metadata

The Metadata section contains information about your app, some of which should match your app's property list. The fields and their explanations are as follows:

- *Version Number*: This should match the Version field from your app's property list. For Graphique, we entered **1.0**.
- *Description*: This is your opportunity to sell your app, explaining to prospective customers what your app does and why they should download it. You have 4,000 bytes, so make them count. Feel free to review the descriptions for other apps on the Mac App Store to give you ideas for what to write. For Graphique, we entered this:

Graphique is a graphing calculator for OS X Lion that was developed in conjunction with the book Beginning OS X Lion Apps Development (Apress 2011). It features a smart, syntax-highlighting editor, results in both graph and table format, a QuickLook plugin, and printing capabilities. Complete source code is available at the Apress site.

Buy the app so you can visualize your equations. Buy the book so you can understand how to build OS X Lion applications!
- *Primary Category*: This should match your app's property list. Category refers to the Mac App Store category to list your app under. For Graphique, we selected Utilities.
- *Secondary Category*: This is an optional field that you can use to specify an additional category. For Graphique, we selected Graphs & Design because of Graphique's ability to display graphs.
- *Keywords*: When users search for apps on the Mac App Store, you want your app to pop up if they're looking for something that your app can do. Be accurate with your keywords so your app appears for all the searches it should, but for none of the searches it shouldn't. For Graphique, we entered **graph, graphing, equation, calculator, sine, cosine, trig**.
- *Copyright*: This should match your app's property list. For Graphique, we entered **Copyright © 2011 Michael Privat and Rob Warner. All rights reserved.**
- *Contact Email Address*: When users have problems with or questions about your app, they e-mail you at this address. For Graphique we entered **graphique@grailbox.com**.

- *Support URL*: The Mac App Store links to this site so users can quickly get help and support for your app. This URL could point to your main web site, or it could point to a specific help forum or documentation site for your app. For Graphique, we entered <http://grailbox.com/graphique>.
- *App URL*: This is an optional URL that points to your app's web site. For Graphique, we entered <http://grailbox.com/graphique>.
- *Privacy Policy URL*: This is an optional URL that points to your privacy policy. For Graphique, we left it blank.
- *Review Notes*: This is an optional field that allows you to communicate information about your app to Apple's app reviewers. The site suggests that, for sandboxed apps, you explain why your app needs the entitlements it's requesting. For Graphique, we entered the following:

Graphique is sandboxed and requests entitlements for reading and saving files and for printing. In the app, you enter equations and see them graphed. You can save the equations to files and later read them back in to Graphique. You can also print your graphs. Graphique also includes a QuickLook plugin so you can see your graphs in Finder.

Rating

Apple will reject apps that it deems objectionable or offensive. The Ratings section gives you an opportunity to disclose how much of any potentially objectionable material appears in your apps, on a scale of None – Infrequent/Mild – Frequent/Intense. Be honest in your assessment. Apple uses this information to help potential customers know the appropriate audience for your apps. For Graphique, we selected None for all categories.

EULA

The EULA section lets you specify your own End User License Agreement (EULA) you want to attach to your app. You can also use the standard EULA for your app by not changing anything in this section. For Graphique, we left this section alone.

Uploads

You must specify at least one screenshot for your app for the Mac App Store entry to display. You can specify multiple screenshots as well. The first screenshot you specify is the one users will see first, although you can change the order of the screenshots after you upload them. Each screenshot should be 1280x800 pixels. For Graphique, we uploaded four screenshots.

When you click Save, you're taken to a screen that identifies your app, as shown in Figure 10-35.

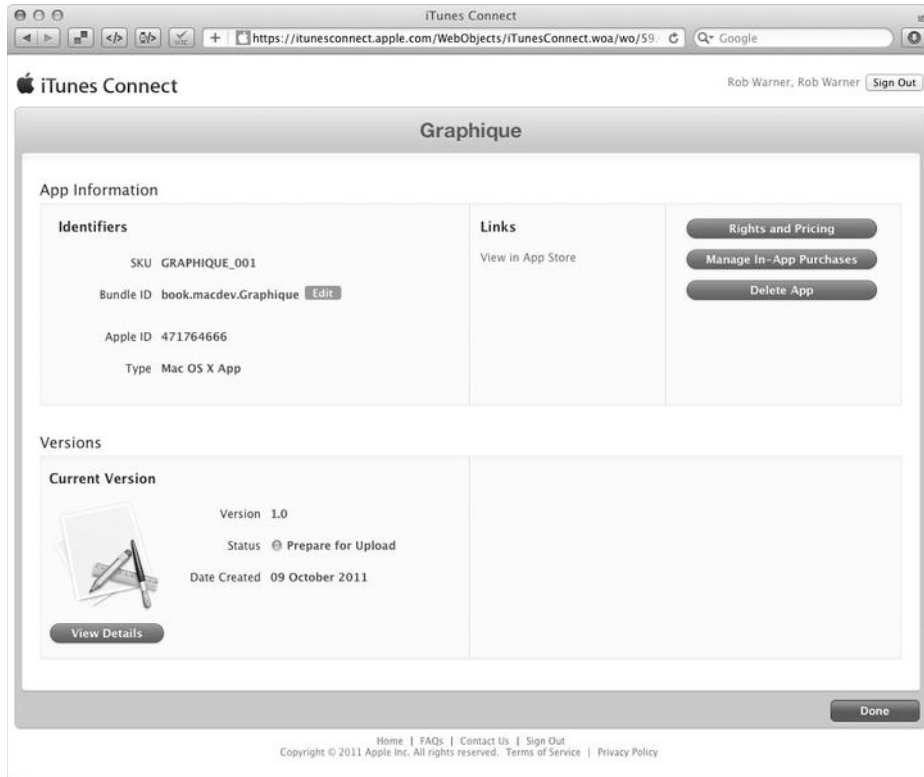


Figure 10-35. The *Graphique* app in iTunes Connect

Click the View Details button to view the information you entered, and click the Ready to Upload Binary button. You are asked whether your app uses cryptography so that Apple can make sure it complies with any export regulations. Answer the question appropriately and click Save.

The next screen talks about having the right Application Loader app, which you have with Xcode 4. Click Continue, and you're returned to your app's information screen, only now the status has changed to Waiting For Upload.

Performing the Upload

Go back to Xcode's Organizer window and select the archive you produced earlier. Click the Validate button, which will ask you for your iTunes Connect login credentials. Enter them and click Next. Xcode connects to iTunes Connect and matches the archive to the app you created in iTunes Connect. Assuming it finds a match, it shows a screen that allows you to confirm that it found the right one, as shown in Figure 10-36.



Figure 10-36. Confirming the application and signing identity

Click Next, and Xcode will validate the archive. You may get an error saying “Unable to extract package metadata.” If you do, you’re not alone. Make no changes and simply try again. You should eventually see a screen like Figure 10-37 that says your app is valid.

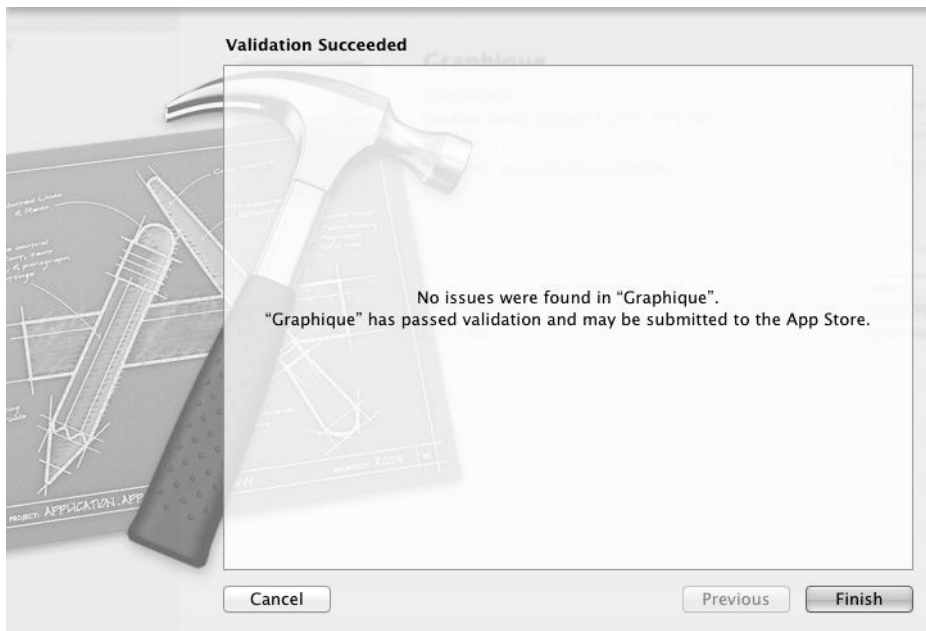


Figure 10-37. Successful validation

Click Finish to return to the Organizer window and, with the same archive still selected, click Submit. You'll be asked for your application record and signing identity, as shown in Figure 10-38. When you click Next, your app is uploaded to Apple. On successful completion, you see a screen like Figure 10-39. Congratulations! Your app is waiting for review.



Figure 10-38. *Selecting your application record and signing identity*

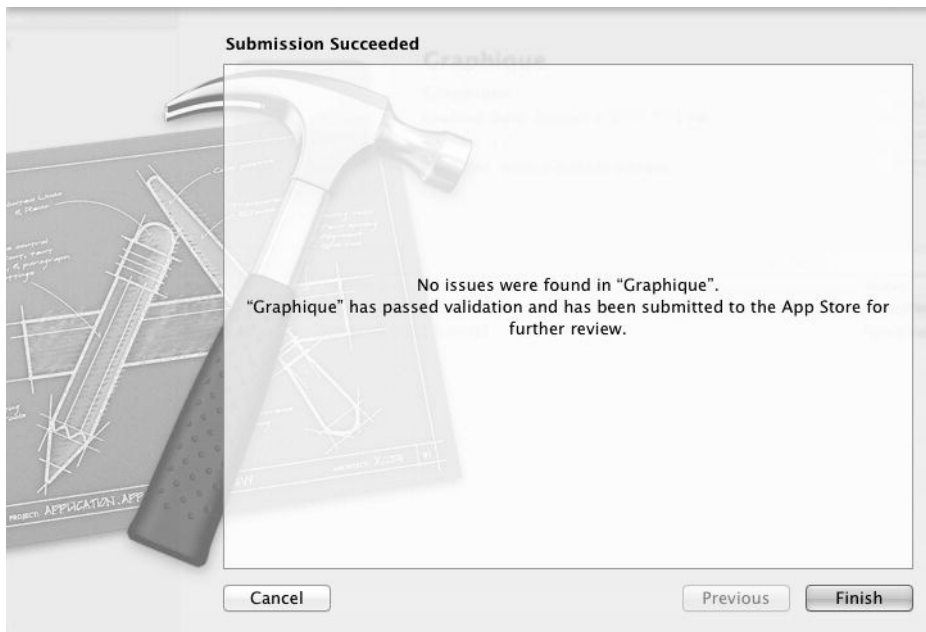


Figure 10-39. *Graphique successfully uploaded*

Here we leave you, waiting on the Mac App Store reviewers to approve your app.

Summary

Submitting an app to the Mac App Store can be a daunting experience as you deal with certificates, App IDs, provisioning profiles, SKUs, and the other information you must provide to land an app in the Mac App Store. Once complete, however, you feel a sense of accomplishment as one of your creations enters the marketplace.

For an app to thrive in the Mac App Store, however, you must continue to nurture it from both a technical and a marketing perspective. Technically, you add features, fix bugs, respond to customer issues, and upload new versions. On the marketing side, you've got to do more than simply expect that since you've built it, people will come. Marketing is a never-ending task that requires diligence, time, and insight. But that's a subject for a different book.

We look forward to seeing your apps on the Mac App Store!

Index

A

- App URL, 359
- Apple's documentation
 - applicationDidFinishLaunching method, 288
 - changeStatusItem: method, 285
 - GraphiqueAppDelegate class, 287
 - initialize: method, 287
 - NSStatusBar, 284
 - PreferencesController.h, 285
 - PreferencesController.m, 285–286
 - showStatusItem outlet, 286
 - updateStatusItemState: method, 285, 287–288
 - windowDidLoad: check box, 285
- AppleWWDRCA.cer file, 337
- applicationDidFinishLaunching: method, 257, 258
- application:openFile: method, 257
- Automatic reference counting (ARC), 49, 50

B

- Bundle ID, 357

C

- COLORS dictionary, 171
- Contact email address, 358
- controlTextDidChange method, 171
- Copyright, 358
- Core Data, 217
 - application process, 218
 - benefits, 218
 - Core Data framework, 219–221
 - data model
 - Add Entity button, 224
 - entities, graph view, 226

- Equation entity, 225
- equations and groups object, 222
- file saving, 223
- graph view, 228
- Group entity, 226
- new data model, 223
- new object model file, 222
- Optional check box, 227
- predefined Core Data attribute types, 224–225
- string representation and timestamp, 225
- To-Many Relationship, 227, 228
- XML file, 221
- definition, 218
- frameworks linked to application, 218–219
- Graphique with equations, 217
- managed object context
 - GraphiqueAppDelegate.h, 230
 - high-level Core Data framework layout, 229
 - initialization process, 229
 - mergedModelFromBundles: method, 230
 - NSManagedObjectContext class, 229
 - NSSQLiteStoreType, 231
 - persistent store, 231
- outline view control
 - Cocoa components, 243
 - double-click prevention, 245
 - equations selection handling, 244, 245
 - NSOutlineViewDelegate, 243
- recently used equations storage

- committing, 235
- equation managed object
 - creation, 235
- EquationItem.h, 238
- EquationItem.m, 238
- GroupItem.h, 239
- GroupItem.m, 239
- initWithNibName:bundle: method, 242
- insertNewObjectForEntityForName:
 - inManagedObjectContext: method, 235
- lazy loading technique, 241
- loadChildrenForItem: method, 240–241
- NSDateFormatter object, 234
- NSFetchRequest object, 234
- outlineView:isItemExpandable:
 - method, 241
- outlineView:numberOfChildrenOfItem:
 - method, 241
- outlineView outlet, 242
- RecentlyUsedEquationsViewController.h, 232–233, 242
- RecentlyUsedEquationsViewController.m, 236–237
- remembering equations method, 233
- remember: method algorithm, 234
- .schema command, 238
- Terminal.app application, 237
- updated equationEntered:
 - method, 237

D

Data graph

- data caching and calculation, 106–108
- GraphTableViewController.xib, 104
- outlets and delegates
 - data source connection, 111
 - Edit GraphTableViewController.h, 109
 - GraphTableViewController.xib, 109
- horizontalSplitView, 108

- NSInteger, 110
- NSTableViewDelegate protocol, 109
 - returning a cell's value, 110
 - table with data, 111, 112
 - Updated draw: Method, 110
 - verticalSplitView, 108
- table columns, 104, 105
- Text Field Cell item, 105
- titled X and Y columns, 106
- Developer Certificate Utility
 - App IDs registration, 335–337
 - Apple's certificate installation, 337
 - certificates creation
 - Create Certificate button, 338
 - development certificate, 338
 - distribution certificates, 338
 - download screen, 343, 344
 - file upload, 342
 - generation, 342, 343
 - installed certificates, 344
 - Keychain Access app, 338–340
 - Mac Installer Package certificate, 344
 - save confirmation, 340, 341
 - “Saved to disk” option, 340, 341
 - provisioning profile
 - App ID, 347
 - creation, 345, 346
 - generation, 347
 - preference pane, 347
 - production, 345, 347
 - screen, 345
 - Xcode Organizer window, 347
 - screen, 334, 335
- dictionaryWithObjectsAndKeys static
 - method, 171

E, F

- End User License Agreement (EULA), 359
- Equation entry panel
 - ARC, 49–50
 - code isolation, 45
 - code reuse, 45
 - custom equation entry component, 48, 49, 52

- equation editor, placeholder string, 57
- Equation Entry View, 52
- GraphiqueAppDelegate.h, 51
- IBOutlet, 50–51
- NSViewController
 - custom components, 46
 - EquationEntryViewController, 46, 47
 - generated files, 48
 - Objective-C class, 46, 47
 - UIViewController class, 46
- placeholder text specification, 56
- split views resizing
 - autosizing schema, 53
 - EquationEntryViewController.xib, 53
 - Graph button, 54, 55
 - Label component, 54
 - springs and struts, 54
 - Text Field component, 55

G

- Graph plotting
 - background painting, 125
 - bounds property, 124
 - curve path, 128
 - drawing function, 124
 - drawRect: method, 124, 126–128
 - horizontal and vertical scaling
 - factors, 124
 - lineToPoint, 128
 - NSBezierPath, 125, 128
 - NSBezierPath class, 124
 - plotted equation, 129
 - scaling factors, 125
- Graphique, 1
 - application architecture, 27–28
 - Mac Dev Center program, 1
 - major components, 23, 24
 - project and targets
 - Graphique.app product, 26
 - Graphique target Build phases, 25
 - GraphiqueTests Build phases, 25, 26
 - Project navigator, 24, 26

- tree structure, 24
- project creation
 - application options, 21
 - application running, 23
 - boilerplate code, 19
 - Cocoa Application, 20
 - code generator, 20
 - common initialization code, 19
 - Git repository, 21
 - new project, 21
 - public static void main method, 19
 - source-control repository, 21
 - system-wide menu bar, 23
- source code and resources
 - application, 28
 - applicationDidFinishLaunching
 - method, 31
 - Application Life Cycle, 32
 - Connections inspector, 30
 - GraphiqueAppDelegate class, 30
 - GraphiqueAppDelegate.m, 32, 33
 - GraphiqueDelegate.h, 32
 - Interface Builder, 29
 - linkages and dependencies, 28
 - main interface, 28
 - MainMenu.xib file, 28, 29
 - nibs, 29
 - NSApplication, 29
 - NSApplicationDelegate method, 34
 - NSApplicationDelegate protocol, 31–33
 - orderFrontStandardAboutPanel
 - method, 34, 35
 - quick documentation, 32
 - Xcode IDE, 29
- Xcode development tools (see Xcode development tools)
- Graphique user interfaces, 119
 - equation editor, 119
 - equation entry field, 119
 - graph view
 - connections, 123
 - drawRect method, 120

- graph plotting (see Graph plotting)
- GraphTableViewController.h, 121
- GraphTableViewController.m file, 122, 123
- GraphTableViewController.xib, 120, 122
- GraphTableViewController's view, 121
- GraphView.h file, 120, 122
- GraphView.m file, 120, 122
- IBOutlet property, 121, 122
- Interface Builder, 120
- NSView class, 120
- setNeedsDisplay method, 123
- smarter equation editor (see Smarter equation editor)
- text and graph toggling (see Tab view)

GraphiqueAppDelegate.m, 324

H

- hasObjects method, 156
- headerEquationToken.h, 142
- Help Book creation
 - description and KEYWORDS, 295
 - directory structure creation, 291–292
 - Entering Equations help page, 295, 296
 - equation_editor_example.png, 295
 - equation_editor.png, 295
 - equations.html, 294
 - files and structure, 291
 - help index creation, 297
 - InfoPlist.strings, 296
 - main help file creation
 - browser window, 293
 - graphique.css, 293
 - GraphiqueHelp.html, 292–293
 - open GraphiqueHelp.html, 293
 - updated help page, 294
 - plist file
 - Graphique.help bundle, 297
 - Info.plist, 298
 - keys, Info.plist, 299–300
 - Xcode Property List editor, 301
 - Xcode's plist editor, 297, 299

- Xcode project, 300
- Help creation, 289
 - Apple's documentation, 290
 - bookmarks, 303–304
 - Entering Equations help, 302, 303
 - Graphique help book, 290, 302
 - Help Book creation (see Help Book creation)
 - help book items, 290
 - misspelled equation, search results, 305, 306
 - resource bundle's directory structure, 290
 - search results, equation, 304, 305
 - user interface, 289

I, J

- iCloud services, 337
- Integrated development environment (IDE), 13
- Interface Builder (IB), 120

K

- Key-value coding (KVC), 113
- Keywords, 358

L

- Last in, first out (LIFO) approach, 155

M

- Mac App Store submission, 323
 - archives, 352–353
 - Graphique termination, 324
 - guidelines, 323
 - icon addition
 - App Icon, 326–327
 - Graphique.icns file, 326
 - Icon Composer with blank window, 325
 - Icon Composer with images added, 325–326
 - image sizes, 324
 - Mac OS X Finder, 324
 - users' Docks, 324
 - initial window size and location
 - Attributes inspector, 334
 - equation field, 331, 332
 - Initial Position option, 332

- MainMenu.xib, 332
- OS X's Autosave feature, 333
- 800 pixels wide window, 332, 333
- Size inspector, 332
- width field, 332, 333
- iTunes Connect account set up, 356–357
- menu clean up
 - About Box, 328–329
 - Full Screen menu item, 330–331
 - Preferences Panel centering, 329
 - unused menu items removal, 330
- property list file, 327
- sandboxing
 - definition, 349–350
 - entitlements, 350–352
 - security measure, 349
- signing code (see Signing code)
- uploading
 - App Name, 357
 - application and signing identity, 361
 - Application Loader app, 360
 - application record and signing identity, 362
 - availability and pricing selection, 357
 - cryptography, 360
 - EULA, 359
 - Metadata, 358–359
 - rating, 359
 - Ready to Upload Binary button, 360
 - screenshots, 359
 - successful completion, 362, 363
 - successful validation, 361
 - “Unable to extract package metadata” error, 361
 - Xcode's Organizer window, 360
- web site
 - artwork, 353, 354
 - creation, 354–356
- Mac Dev Center, 2
- Mac OS X desktop, 247
 - blank generic document icon, 260, 261
 - default preview, 260
 - file types registration, Lion
 - Graphique editor, 255–256
 - Graphique equation files, 257–259
 - .graphique file type, 253
 - UTI, .graphique extension, 253–255
 - Graphique XML files
 - Apple's PropertyList DTD, 251
 - destination file selection, 250
 - equation entry view controller, 249
 - Equation object, 250
 - GraphiqueAppDelegate.m method, 252
 - loadData method, 252, 253
 - MainMenu.xib file, 252
 - Microsoft Word, 247
 - NSDictionary objects, 250
 - NSOpenPanel, 252
 - NSSavePanel, 251
 - openDocument method, 252
 - property list files, 247
 - Save and Save As menu items, 248
 - Save As method, 248–249
 - saveDocumentAs method, 250
 - text variable, 250
 - menu bar items
 - Apple's documentation (see Apple's documentation)
 - application functions, 277
 - NSStatusBar and NSStatusItem, 277–278
 - status item (see Status item)
 - third-party applications, 277
 - pictures/movies, thumbnail, 259
 - plug-in testing
 - Arguments Passed on Launch section, 268, 269
 - debug preview, 270, 271
 - GraphiqueQL scheme, 270
 - QL plug-in build phases, 269, 270
 - qlmanage link, 268
 - setup process, 269

- Terminal window, 267
- preview implementation
 - Cocoa.framework, 266, 267
 - export: method, 264
 - GeneratePreviewForURL
 - function, 266
 - saveDocumentAs: method, 265
 - “Quick Look”, definition, 259
- Quick Look plug-in
 - Add Copy Files, 276
 - application bundle, 276, 277
 - Copy Files Build Phase, 275
 - document content type UTIs, 264
 - GenerateThumbnailForURL and
 - GeneratePreviewForURL, 264
 - GraphiqueQL, 262, 263, 275, 276
 - /Library/QuickLook, 261
 - Mac App Store, 276
 - options, 262, 263
 - target selection, 261, 262
- thumbnail implementation
 - Finder window, 274, 275
 - GenerateThumbnailForURL
 - function, 271
 - qlmanage argument, 272, 273
 - Quick Look thumbnail testing,
 - 273, 274
- Mac OS X Lion, 1
- Model-View-Controller (MVC) design
 - pattern, 85
- Mutable attributed string, 171

N, O

- newTokenFromString method, 143–144
- NSApplicationDelegate, 28
- NSAttributedString class, 137
- NSBackgroundColorAttributeName, 171
- NSBaselineOffsetAttributeName
 - attribute, 173
- NSColor class, 125
- NSColor value, 171
- NSForegroundColorAttributeName, 171
- NSMutableArray class, 157
- NSRectFill function, 125
- numberWithInt method, 171

P

- Primary category, 358
- Privacy policy URL, 359
- public.content type, 254–255
- public.data type, 255

Q

- QuickLook plug-in, 349

R

- Review notes, 359
- RGBA colors, 125

S

- Secondary category, 358
- Signing code
 - build configuration, 348–349
 - Developer Certificate Utility: (see
 - Developer Certificate Utility)
- SKU number, 357
- Smarter equation editor
 - equation colorizing, 171–173
 - equation entry field
 - attributed string, 137
 - NSAttributedString class, 137
 - NSControl instance, 137
 - NSTextField widget, 137
 - Rich Text, 137, 138
- EquationEntryViewController object,
 - 170
- evaluator updation
 - awk, 178
 - description method, 180
 - Equation.m, 180
 - evaluateForX method, 180–181
 - expand method, 178–180
 - implicit exponents, implicit
 - multiplication, and pi, 177, 181
 - symbols, 177
 - x^2 graph, 177
 - x^2 graph, 177, 178
- exponent superscripting, 170
 - background color, 173
 - controlTextDidChange method,
 - 173–175
 - NSFontAttributeName attribute,
 - 173

- NSSuperscriptAttributeName
 - attribute, 173
 - implicit exponents, 137
 - implicit multiplication, 136, 137
 - inline error highlighting, 136
 - parenthesis matching, 136
 - parsing, equation
 - arrays in Equation.m, 140
 - Equation.h with a tokens
 - property, 141
 - Equation.m, 141
 - EquationTokenizer class, 140
 - initialize: method, 140–141
 - initializing tokens, 142
 - initWithString: method, 141, 142
 - OPERATORS array, 140
 - SYMBOLS array, 141
 - TRIG_FUNCTIONS array, 140
 - setting colors, 170–171
 - superscript exponents, 137
 - symbols, 137
 - syntax coloring, 136
 - syntax highlighting, 169–170
 - text field, 136
 - tokenize method (see Tokenize
 - method)
 - tokens
 - close parentheses, 138
 - EquationToken class, 139
 - EquationToken.h, 139
 - EquationToken.m, 139–140
 - exponents, 138
 - implicit multiplication, 138
 - invalid input, 138
 - NSObject. EquationToken.h, 139
 - numbers, 138
 - open parentheses, 138
 - operators, 138
 - spaces, 138
 - symbols, 138
 - syntax coloring, 138
 - trigonometric functions, 138
 - variables, 138
 - trigonometric functions, 136
 - validator updation
 - Graphique tests, 177
 - invalid input, 177
 - invalid parentheses, 175
 - invalid trigonometric functions, 175
 - rules, 175
 - validate method, 176–177
 - stack NSMutableArray, 156
 - Status item
 - applicationDidFinishLaunching:
 - method, 283
 - configureStatusItem: method, 283
 - equations, 284
 - GraphiqueAppDelegate.h, 282–283
 - graphique18.png, 279
 - Graphique status item icon, zoomed
 - in, 278, 279
 - GraphiqueStatusItemMenuDelegate.
 - h, 279–280
 - GraphiqueStatusItemMenuDelegate.
 - m, 280, 282
 - GroupItem instance, 280
 - Images directory, 279
 - loadData: and
 - showEquationFromString:
 - methods, 284
 - menuNeedsUpdate: method, 280
 - RecentlyUsedEquationsViewController.h, 283
 - showEquationFromString: method, 282
 - statusMenuItemSelected: method, 280
 - Support URL, 359
 - Syntax-colored equation, 173
- ## T
- Tab view
 - addition, 131
 - attributes inspector, 131
 - controller switching
 - connections, 134
 - tabs showing Graph View, 135
 - tabs showing table view, 136
 - expanded structure, 131
 - Fill Container Horizontally, 132, 133
 - Fill Container Vertically, 132, 133
 - GraphTableViewController.xib, 130

- graph views maximization, 130
- new tab structure with components, 132
- Object Library, 130
- Objects panel, 130, 132
- rename, 134
- resize, 133
- Size inspector tab, 132
- Tokenize method
 - Equation.m, 142
 - exponents recognition
 - Command+U, 155
 - ExponentTests class, 152, 153
 - ExponentTests.h, 154
 - ExponentTests.m, 154–155
 - finding exponents, 150, 152
 - GraphiqueTests folder, 152
 - rules, 150
 - grouping spaces, 146–148
 - initial method, 142
 - multiple decimal points detection, 161–163
 - recognizing numbers, 145–146
 - stack, parenthesis matching, 155
 - balancing, 158–161
 - creation, 156–157
 - LIFO approach, 155
 - NSMutableArray class, 155
 - popping, 155
 - pushing, 155
 - testing, 157–158
 - string–token conversion, 143–145
 - testing (see Tokenizer testing)
 - trigonometric functions and symbols
 - detection, 148, 150
 - EquationToken object, 150
 - lowercaseString method, 148
 - newTokenFromString, 150
 - NSMakeRange function, 148
 - substringWithRange method, 148
 - SYMBOLS array, 148
 - TRIG_FUNCTIONS array, 148
- Tokenizer testing
 - EquationTokenizeTests class, 164
 - EquationTokenizeTests.h, 164
 - EquationTokenizeTests.m, 164

- exponents, 165
- GraphiqueTests folder, 164
- invalid cases, 169
- parenthesis matching, 168–169
- simple equations, 165
- trigonometric functions and symbols, 167
- whitespace, 166

U

- UNIX Development tools, 6
- User input handling, 73
 - Alert Window display, 99–100
 - App Store comments, 91
 - button presses
 - Edit Equation.h, 88
 - EquationEntryViewController.h, 90
 - Equation.h, 88
 - evaluateForX: method, 89–90
 - GraphTableViewController
 - method, 90, 91
 - IBAction, 85–87
 - init method, Equation.m, 88
 - MVC design pattern, 85
 - NSString, 87
 - Return key, 85
 - UI components, 85
 - data graph (see Data graph)
 - Graphique application, 73, 112
 - KVC, 113
 - Model Key Path field, 114
 - parabola, 0.1 interval, 117
 - parabola, 5.0 interval, 116
 - real-time validation
 - catch notifications, 101, 102
 - feedback attribute, UI
 - component, 103
 - feedback label, 100
 - real-time validation error, 104
 - text field delegate, interface
 - builder, 103
 - Wrapping Label object, 102
 - slider bound, 1.0 interval, 115
 - split view sizes
 - maximum size constraining
 - method, 77–79

- minimum size constraining
 - method, 76
 - NSSplitViewDelegate protocol, 76
 - subview collapse, 83–84
 - unit testing
 - 102 and 103 error codes, 98–99
 - definition, 94
 - GraphiqueTests class, 95
 - STFail() function, 95–97
 - unit test failures, 97, 98
 - unit test launcher, 97
 - window resizing
 - Attributes inspector, 74
 - Equation Entry View, 79, 80
 - Equation Entry Views subviews
 - size adjustment, 80
 - Graphique stretches, 82
 - Graphique window, 76
 - open MainMenu.xib, 79
 - Recent Equations View, 100
 - pixels, 82, 83
 - Recent Equations View Shrunk, 82
 - Resize check box, 74
 - splitView:resizeSubviewsWithOldSize:, 81
 - writing validator
 - equation validation method and private category, 93–94
 - Graphique error codes, 92
 - NSError object, 92
 - romanNumeralValue: to NSNumber method, 92
 - validate: method, 92
 - User interface, 37
 - equation entry panel (see Equation entry panel)
 - graph panel, 57
 - empty Graph View, 58
 - Graphique application, 61, 62
 - GraphTableViewController, 57, 61
 - horizontal slider, 58–60
 - table view, 60
 - Graphique application, 38
 - horizontal NSSplitView
 - Document Outline, 39, 40
 - Graphique application, 43
 - Horizontal Split View object, 41
 - MainMenu.xib, 39
 - Size inspector, 42
 - View object, 40
 - layout, 39
 - recently used equations
 - adding a toolbar, 71–72
 - data source creation, 67–69
 - data source creation data display, 69–71
 - EquationItem.h, 65
 - EquationItem.m, 65
 - GraphiqueAppDelegate.h, 64
 - GraphiqueAppDelegate.m, 64
 - GroupItem.h, 66
 - GroupItem.m, 66
 - Mac OS X implementation, 62
 - NSOutlineViewDataSource protocol, 65
 - numberOfChildren method, 67
 - Outline View object, 63
 - RecentlyUsedEquationsViewController files, 62, 63
 - vertical NSSplitView, 43–45
 - widgets, 37
 - UTExportedTypeDeclarations key, 253
- V**
- Version number, 358
- W**
- WWDR Intermediate Certificate, 337
- X, Y, Z**
- Xcode development tools
 - App Store installation
 - Apple ID and password, 9
 - downloading in Launchpad, 9
 - 3GB disk space, 12
 - Install Xcode app, 10
 - iOS SDK agreements, 11
 - password, 11
 - writing files to drive, 11, 12
 - Xcode Installer splash screen, 10
 - components, 15, 16

- “Create a new Xcode project”
 - option, 13
- Debug Area and Debug Bar, 17
- Editor Area and Jump Bar, 16
- Editor Selector, 18
- Filter Bar, 17
- IDE, 13
- Inspector Pane, 18
- Inspector Selector Bar, 18
- launch, 12, 13
- Library Pane, 18
- main.m file, 15
- Navigator Area and Navigator
 - Selector Bar, 17
- project name and type, 13, 14
- template selection, 13, 14
- Toolbar, 18–19
- tooltip display, 16

- View Selector, 18
- Visual Studio levels, 2
- Web download installation
 - Change Install Location, 6, 7
 - determination, 3, 4
 - disk image file, 3
 - instructions, About Xcode file, 3
 - license agreement, 5, 6
 - options, 6, 7
 - password, 6, 8
 - preparing to install, 3, 4
 - successful completion, 9
 - System Tools, 6
 - UNIX Development tools, 6
 - writing Xcode files to drive, 8
- window with HelloWorld project, 13, 15
- Xcode 4 software, 2