MOBILE PROGRAMMING SERIES

iOS AUTO LAYOUT DEMYSTIFIED

ERICA SADUN

FREE SAMPLE CHAPTER

iOS Auto Layout Demystified

Erica Sadun

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

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

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

International Sales
international@pearsoned.com

Visit us on the Web: http://informit.com/aw

ISBN-13: 978-0-13-344065-2
ISBN-10: 0-13-344065-6

Editor-in-Chief
Mark Taub

Senior Acquisitions Editor
Trina MacDonald

Senior Development Editor
Chris Zahn

Managing Editor
Kristy Hart

Project Editor
Jovana Shirley

Copy Editor
Keith Cline

Proofreader
Sheri Cain

Technical Reviewers
Richard Wardell

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
Nonie Ratcliff

❖

Hop. Hop. THOOM.

❖

Contents at a Glance

Acknowledgments

No book is the work of one person. I want to thank my team who made this possible. The lovely Trina MacDonald green lit this title, thus ultimately providing the opportunity you now have to read it. Chris Zahn is my wonderful development editor, and Olivia Basegio makes everything work even when things go wrong.

I send my thanks to the entire Addison-Wesley/Pearson production team, specifically Kristy Hart, Jovana San Nicolas-Shirley, Keith Cline, Sheri Cain, Nonie Ratcliff, and Chuti Prasertsith.

Thanks go as well to Neil Salkind, my agent of many years; to Rich Wardwell, my technical editor; and to my colleagues, both present and former, at TUAW and the other blogs I've worked at.

I am deeply indebted to the wide community of iOS developers who supported me in IRC and who helped by reading drafts of this book and offering feedback. Particular thanks go to Oliver Drobnik, Aaron Basil (of Ethervision), Harsh Trivedi, Michael Prenez-Isbell, Alex Hertzog, Neil Taylor, Maurice Sharp, Rod Strougo, Chris Samuels, Hamish Allan, Jeremy Tregunna, Lutz Bendlin, Mahipal Raythattha, Robert Jen, Greg Hartstein, Jonathan Thompson, Ajay Gautam, Shane Zatezalo, Wil Macaulay, Bill DeMuro, Evan Stone, Alex Mault, David Smith, Duncan Champney, August Joki, Remy "psy" Demarest, Joshua Weinburg, Emanuele Vulcano, and Charles Choi. Their techniques, suggestions, and feedback helped make this book possible. If I have overlooked anyone who contributed to this effort, please accept my apologies for the oversight.

Special thanks also go to my husband and kids. You are wonderful.

About the Author

Erica Sadun is the bestselling author, coauthor, and contributor to several dozen books on programming, digital video and photography, and web design, including the widely popular *The Core iOS 6 Developer's Cookbook, Fourth Edition.* She currently blogs at TUAW.com, and has blogged in the past at O'Reilly's Mac Devcenter, Lifehacker, and Ars Technica. In addition to being the author of dozens of iOS-native applications, Erica holds a Ph.D. in computer science from Georgia Tech's Graphics, Visualization and Usability Center. A geek, a programmer, and an author, she's never met a gadget she didn't love. When not writing, she and her geek husband parent three geeks-in-training, who regard their parents with restrained bemusement when they're not busy rewiring the house or plotting global domination.

Preface

Auto Layout re-imagines the way developers create user interfaces. It creates a flexible and powerful system that describes how views and their content relate to each other and to the windows and superviews they occupy. In contrast to older design approaches, this technology offers incredible control over layout with a wider range of customization than frames, springs, and struts can express. Somewhat maligned by exasperated developers, Auto Layout has gained a reputation for difficulty and frustration, particularly when used through Interface Builder (IB).

That's why this book exists. You're about to learn Auto Layout mastery by example, with plenty of explanations and tips. Instead of struggling with class documentation, you'll learn in simple steps how the system works and why it's far more powerful than you first imagined. You'll read about common design scenarios and discover best practices that make Auto Layout a pleasure rather than a chore to use.

This book aims to be inspirational. I've tried to show examples of nonobvious ways to use Auto Layout to build interactive elements, animations, and other features beyond what you might normally encounter in IB. These chapters provide a launch pad for Auto Layout work and introduce unfamiliar features that expand your design possibilities.

As the name suggests, this book is primarily targeted at iOS developers. I have included OS X coverage where possible. So, if you're an OS X developer, you're not left out completely in the cold. I live primarily in the iOS world. Please keep that in mind as you read.

Auto Layout has made a profound difference in my day-to-day development. I wrote this book hoping it will do the same for you. It's my intention that you walk away from this book with a solid grounding in Auto Layout. And, if I'm lucky, the book will provide you with a "Eureka!" moment or two to lead you forward.

—Erica Sadun, February 2013

How This Book Is Organized

This book offers practical Auto Layout tutorials and how-tos. Here's a rundown of what you find in this book's chapters:

- **Chapter 1, "Introducing Auto Layout"**—Ready to get started? This chapter explains the basic concepts that lie behind Auto Layout and, for those who need to, how to opt out of Auto Layout until you're ready to proactively take advantage of its many benefits.

- **Chapter 2, "Constraints"**—With Auto Layout, you build interfaces by declaring rules about views. Each layout rule you add creates a requirement about how part of the interface should be laid out. These rules are ranked based on a numeric priority that you supply to the system, and Auto Layout builds your interface's visual presentation accordingly. This chapter introduces constraints, the rules of your layout, and explains why those rules must be unambiguous and satisfiable.
- **Chapter 3, "Interface Builder Layout"**—Working with constraint-based design in Interface Builder can sometimes prove a frustrating experience for developers new to Auto Layout. With IB, you must move away from a normal developer mindset. You can't approach IB-based Auto Layout by asking, "Which constraints do I want to add to express my design goals?" Instead, you need to consider, "How do I modify the constraints that I've built and that Interface Builder has given me to achieve my design goals?" This chapter explains how.
- **Chapter 4, "Visual Formats"**—This chapter explores what visual constraints look like, how you build them, and how they are used in your projects. You'll read how metrics dictionaries and constraint options extend visual formats for more flexibility. And you'll see numerous examples that demonstrate these formats and explore the results they create.
- **Chapter 5, "Debugging Constraints"**—Constraints can be maddeningly opaque. The code and interface files you create them with don't lend themselves to easy perusal. It takes only a few "helpful" Xcode log messages to make some developers start tearing out their hair. This chapter dedicates itself to shining light upon the lowly constraint and helping you debug your work.
- **Chapter 6, "Building with Auto Layout"**—Designing for Auto Layout changes the way you build interfaces. It's a descriptive system that steps away from exact metrics like frames and centers. You focus on expressing relationships between views, describing how items follow one another onscreen. You uncover the natural relationships in your design and detail them through constraint-based rules. This chapter introduces the expressiveness of Auto Layout design, spotlighting its underlying philosophy and offering examples that showcase its features.
- **Chapter 7, "Layout Solutions"**—The chapters that led up to this focused on know-how and philosophy. This chapter introduces solutions. You'll read about a variety of real-world challenges and how Auto Layout provides practical answers for day-to-day development work. The topics are grab bag, showcasing requests developers commonly ask about.

About the Sample Code

This book follows the trend I started in my iOS Developer Cookbooks. This book's iOS sample code always starts off from a single main.m file, where you'll find the heart of the application powering the example. This is not how people normally develop iOS or Cocoa applications, or, honestly, how they should be developing them, but it provides a great way

of presenting a single big idea. It's hard to tell a story when readers must search through many files while trying to find out what is relevant and what is not. Offering a single launching point concentrates that story, allowing access to that idea in a single chunk.

The presentation in this book does not produce code in a standard day-to-day best-practices approach. Instead, it offers concise solutions that you can incorporate back into your work as needed. For the most part, the examples for this book use a single application identifier: com.sadun.helloworld. This avoids clogging up your iOS devices with dozens of examples at once. Each example replaces the preceding one, ensuring that your home screen remains relatively uncluttered. If you want to install several examples simultaneously, simply edit the identifier, adding a unique suffix, such as com.sadun.helloworld.table-edits.

You can also edit the custom display name to make the apps visually distinct. Your Team Provisioning Profile matches every application identifier, including com.sadun.helloworld. This allows you to install compiled code to devices without having to change the identifier; just make sure to update your signing identity in each project's build settings.

There is a smattering of OS X code in here as well. This is not (as the title suggests) an OS X-centered book, but I've covered OS X topics where it made sense to do so. I spend the majority of my time in iOS, so please forgive any OS X faux pas I make along the way and do drop me notes to help me correct whatever I got wrong.

Getting the Sample Code

You'll find the source code for this book at http://github.com/erica/Auto-Layout-Demystified on the open-source GitHub hosting site. There, you find a chapter-by-chapter collection of source code that provides working examples of the material covered in this book.

If you do not feel comfortable using git directly, GitHub offers a download button. It was at the right-center of the page at the time of this writing. It enables you to retrieve the entire repository as a ZIP archive or tarball.

Contribute!

Sample code is never a fixed target. It continues to evolve as Apple updates its SDK and the Cocoa Touch libraries. Get involved. You can pitch in by suggesting bug fixes and corrections and by expanding the code that's on offer. GitHub allows you to fork repositories and grow them with your own tweaks and features and then share those back to the main repository. If you come up with a new idea or approach, let me know. My team and I are happy to include great suggestions both at the repository and in the next edition of this book.

Getting Git

You can download this book's source code using the git version control system. An OS X implementation of git is available at http://code.google.com/p/git-osx-installer. OS X git

implementations include both command-line and GUI solutions, so hunt around for the version that best suits your development needs.

Getting GitHub

GitHub (http://github.com) is the largest git-hosting site, with more than 150,000 public repositories. It provides both free hosting for public projects and paid options for private projects. With a custom Web interface that includes wiki hosting, issue tracking, and an emphasis on social networking of project developers, it's a great place to find new code or collaborate on existing libraries. You can sign up for a free account at their Web site, which then allows you to copy and modify this repository or create your own open-source iOS projects to share with others.

Contacting the Author

If you have any comments or questions about this book, please drop me an e-mail message at erica@ericasadun.com, or stop by the GitHub repository and contact me there.

Editor's Note: We Want to Hear from You!

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

You can e-mail or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

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

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

E-mail: trina.macdonald@pearson.com
Mail: Trina MacDonald
Senior Acquisitions Editor
Addison-Wesley/Pearson Education, Inc.
75 Arlington St., Ste. 300
Boston, MA 02116

1

Introducing Auto Layout

Auto Layout re-imagines the way developers create user interfaces. It's provides a flexible and powerful system that describes how views and their content relate to each other and to the windows and superviews they occupy. In contrast to older design approaches, this technology offers incredible control over layout with a wider range of customization than frames, springs, and struts can express. Somewhat maligned by exasperated developers, Auto Layout has gained a reputation for difficulty and frustration, particularly when used through Interface Builder (IB).

That's why this book exists. You're about to learn Auto Layout mastery by example, with plenty of explanations and tips. Instead of struggling with class documentation, you'll learn, in simple steps, how the system works and why it's far more powerful than you first imagined. You'll read about common design scenarios and discover best practices that make Auto Layout a pleasure rather than a chore to use.

Ready to get started? This chapter explains the basic concepts that lie behind Auto Layout and, for those who need to, how to opt out of Auto Layout until you're ready to proactively take advantage of its many benefits.

> **Note**
>
> Auto Layout first debuted for iOS 6 in 2012. It also appears in OS X starting with 10.7 Lion. Its technology is based on the Cassowary constraint-solving toolkit developed at the University of Washington by Greg J. Badros and Alan Borning. Cassowary's constraint system has been ported to JavaScript, .NET/Java, Python, Smalltalk, C++, and to Cocoa.
>
> The similarly themed `CAConstraint` and `CAConstraintLayoutManager` classes debuted in OS X 10.5 and are used with Core Animation layer trees. They are otherwise unrelated to Auto Layout.

Saying “No” to Auto Layout

Before diving into Auto Layout, it helps to know how to opt out. If you’re not yet ready to use these new features, you can easily switch off Auto Layout for individual storyboard and NIB files. All it takes is a simple switch to have Xcode revert you to the old style of Autosizing edits.

Here’s what you do:

1. In Xcode, select any user interface document (storyboards or NIB file) from the Project Navigator (View > Navigators > Show Project Navigator).
2. Open the File Inspector (View > Utilities > Show File Inspector).
3. In the File Inspector, locate the Interface Builder Document section. Just below Document Versioning, you’ll find a Use Autolayout (or Use Auto Layout) check box, which you see in Figure 1-1.

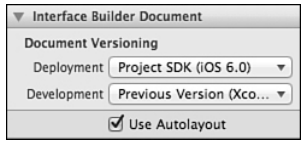

Figure 1-1 Disable Auto Layout by unchecking the Use Autolayout (sic) or Use Auto Layout box in Xcode’s File Inspector. This option appears for both iOS and OS X projects. Although Apple documentation from 2012 and later now universally refers to the technology as Auto Layout, some references still use *Autolayout* as a single phrase.

4. Uncheck this box to return Interface Builder to Autosizing behavior.

Under Autosizing, views use the `autoresizingMask` property to ensure that they resize correctly, such as when a device reorients or a user resizes a window. With Auto Layout disabled, you work with struts, springs, and `autoresizingMask`s. Figure 1-2 shows the Size Inspector (View > Utilities > Show Size Inspector) for a view with Auto Layout enabled (left) and disabled (right).

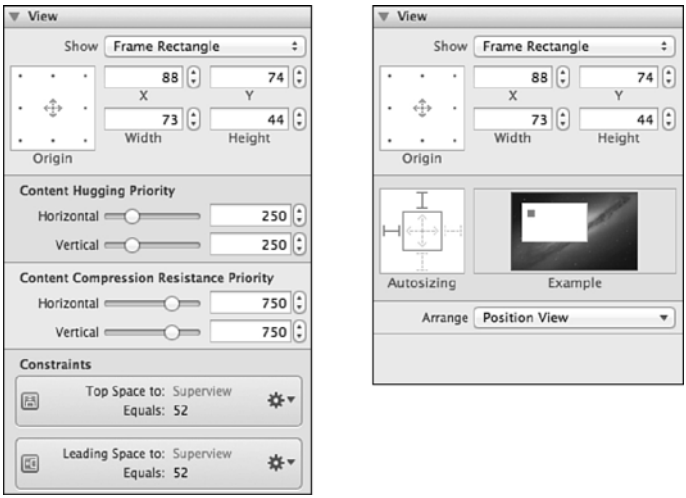

Figure 1-2 When Auto Layout is enabled (left), IB's Size Inspector offers controls to adjust a view's layout priorities and a provides list of constraints that mention the view. When disabled (right), the inspector reverts to the Autosizing struts and springs editor that correspond to a view's `autoresizingMask`.

In code, Autosizing layout is even easier. You opt views *out* of Autosizing to participate in the new Auto Layout system. All views default to the older behavior even when modern runtimes use constraint-based layout. That's because the runtime invisibly handles the translation between your old-style code and the new-style constraint system on your behalf.

When a view's `translatesAutoresizingMaskIntoConstraints` property is set to `YES` (the default), the runtime uses that view's autoresizing mask to produce matching constraints in the new Auto Layout system. The rules apply like they always did, even though the implementation details have been modernized.

Here's what Apple has to say on the matter. This comment is from the UIView.h header file, and it's essentially identical to the text in NSLayoutConstraint.h on OS X:

> By default, the autoresizing mask on a view gives rise to constraints that fully determine the view's position. Any constraints you set on the view are likely to conflict with autoresizing constraints, so you must turn off this property first. IB will turn it off for you.

Autoresizing translation creates constraints to implement Autosizing rules and adds them to each superview. As you will see throughout this book, this pattern of generating constraints and adding them to a superview is quite common. In Chapter 2, "Constraints," you learn exactly where each constraint should live. In this instance, you do no work yourself. The Auto Layout system handles all migration from autoresizing.

When you're ready to move to Auto Layout, make sure that the constraints automatically generated for you don't conflict with constraints you build on your own. Auto Layout constraints belong to the private `NSAutoresizingMaskLayoutConstraint` class. Constraints you build belong to the public `NSLayoutConstraint` class.

You may see instances of the resizing constraints mentioned in debugging logs produced by Xcode and should not be surprised by them. When conflicts do occur, check each view's `translatesAutoresizingMaskIntoConstraints` property. Ensure that the Auto Layout system is not trying to create constraints that conflict with the ones that you are building yourself. In many cases, disabling autoresizing mask translation resolves basic conflicts.

That's not to say that you must disable autoresizing entirely. You are welcome to mix and match autoresizing views with constraint-based layout as long as rules don't clash. For example, you can load a NIB whose subviews are laid out using struts and springs and allow that view, in turn, to operate as a first-class member of the Auto Layout world (see Chapter 3, "Interface Builder Layout," for an example). The key is encapsulation. As long as rules do not conflict, you can reuse complex views you have already established in your projects.

Saying "Yes" to Auto Layout

Auto Layout revolutionizes view layout with something wonderful, fresh, and new. Apple's layout features make your life easier and your interfaces more consistent, and they add resolution-independent placement for free. You get all this regardless of device geometry, orientation, and window size.

Auto Layout works by creating relationships between onscreen objects. It specifies the way the runtime system automatically arranges your views. The outcome is a set of robust rules that adapt to screen and window geometry. With Auto Layout, you describe requirements (called *constraints*) that specify how views relate to one another and you set view properties that describe a view's relationship to its content. With Auto Layout, you can make requests such as the following:

- Match one view's size to another view's size so that they always remain the same width.
- Center a view in its parent no matter how much the parent reshapes.
- Align one view to another view's bottom while laying out a row of items.
- Offset a pair of items by some constant distance (for example, pad with a standard 8-point space).
- Tie the bottom of one view to another view's top so that when you move one, you move them both.
- Don't allow an image view to shrink to the point where the image cannot be fully seen at its natural size. (That is, don't compress or clip the view's content.)
- Prevent a button from showing too much padding around its text.

The first five items in this list describe constraints that define view geometry and layout, establishing visual relationships between views. The last two items reference view properties that relate each view to the content it presents. When working with Auto Layout, you negotiate both of these kinds of tasks.

Visual Relationships

Figure 1-3 shows a custom iOS control built entirely with Auto Layout. This picker enables users to select a color. Each pencil consists of a fixed-size tip view placed directly above a stretchable bottom view. As users make selections, items move up and down to indicate their current choice. Auto Layout constraints ensure that the each tip stays exactly on top of its base, that each "pencil" is sized to match, and that the paired tip/base items are laid out in a bottom-aligned row.

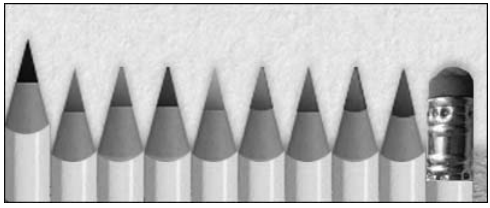

Figure 1-3 This pencil-picker custom control was built entirely with Auto Layout.

This particular pencil picker is built programmatically; that is, a data source supplies the number of pencils and the art for each tip. By describing the relationships between each item, Auto Layout simplifies the process of extending this control. You need only say "place each new item to the right, match its width to the existing pencils, and align its bottom" to grow this picker from 10 items to 11, 12, or more. Best of all, constraint changes can be animated. The pencil tip animates up and down as the base reshapes to new constraint offsets.

Content-Driven Layout

Auto Layout can also consider a view's content during layout. To accomplish this, it may need to negotiate competing requests. For example, imagine a resizable content view with several subviews, like the one shown in Figure 1-4. Suppose that you want to be able to resize this view but don't want to clip any subview content while doing so. Auto Layout helps you express these desires and rank them so that the system makes sure not to clip when resizing.

Figure 1-4 shows a small OS X application whose primary window protects the content of its two subviews. These include a label whose content is the string Label and a resizable button whose content is, similarly, the string Button. The original content view as the application launches is shown in the left screenshot.

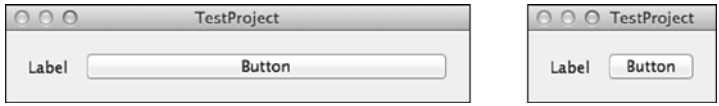

Figure 1-4 Auto Layout can ensure that the stretchable button shown in the original view (left) won't clip its subviews while resizing. The window cannot resize any smaller than the small view (right) because doing so would cause either the label or button to clip.

At the right of Figure 1-4, you see the smallest possible version of this view. Because its Auto Layout rules resist clipping (these rules are called *compression resistance*), the window cannot resize any further. The only way to allow it to shrink beyond this size is to demote one or both of its "do not clip" subview rules.

A similar rule called *content hugging* allows a view to resist padding and stretching, keeping the frame of each view close to the natural size of the content it presents. Both compression resistance and content hugging are demonstrated further later in this chapter as well as in Chapter 2.

Prioritizing Rules

Rule-balancing forms an important backbone of Auto Layout design work. You not only specify the layout qualities of each view, you also prioritize them. When rules come into conflict—and they do quite regularly—the system uses these rankings to select the most important layout qualities to preserve.

In the example of Figure 1-4, the integrities of the label and button contents have priority over any request for a smaller window. That forces a natural minimum on the window size, preventing it from resizing any further than the smaller version shown at the bottom of the figure.

Auto Layout Strengths

To summarize, you can do a lot with Auto Layout. Auto Layout offers a flexible and powerful descriptive system that updates and strengthens view layout. You can make rules about how views relate to each other and to their content, and you can prioritize rules to determine which rules prevail in the event of conflict.

You've seen just a taste of what Auto Layout can do. If you're not using Auto Layout, you're missing out on one of the best tools Apple has delivered to developers in years.

> **Note**
>
> Auto Layout does not exist on OS versions earlier than iOS 6 and OS X 10.7. Should you attempt to run on earlier systems, your app will crash. To write backward-compatible applications that leverage Auto Layout, make sure to implement two versions of your layout routines: one for Auto Layout and one with Autosizing for earlier deployments. If you use Autosizing exclusively, you will miss out on the power of an amazing new technology.

Constraints

Constraints are rules that allow you to describe view layout. They limit how things relate to each other, specifying how they may be laid out. With constraints, you can say "these items are always lined up in a horizontal row" or "this item resizes itself to match the height of that item." Constraints provide a layout language that you add to views to describe visual relationships.

The constraints you work with belong to the `NSLayoutConstraint` class. This Objective-C class specifies relationships between view attributes, such as heights, widths, positions, and centers. What's more, constraints are not limited to equalities. They can describe views using greater-than-or-equal and less-than-or-equal relations so that you can say that one view must be at least as big or no bigger than another. Auto Layout development is built around creating and adjusting these relationship rules in a way that fully defines your interfaces.

Together, an interface's constraints describe the ways views can be laid out to dynamically fit any screen or window geometry. In Cocoa and Cocoa Touch, a well-designed interface consists of constraints that are *satisfiable* and *sufficient.* The next sections introduce these terms and explain why they are important.

> **Note**
>
> Each individual constraint refers to either one or two views. Constraints relate one view's attributes either to itself or to another view.

Satisfiability

Cocoa/Cocoa Touch takes charge of meeting layout demands via a constraint satisfaction system. The rules must make sense. In logic systems, this is called *satisfiability* or *validity.* A view cannot be both to the left *and* the right of another view. So, the key challenge when working with constraints is ensuring that the rules are rigorously consistent.

Any views you lay out in Interface Builder are guaranteed to be satisfiable. You cannot create a *wrong* interface with inconsistent rules in IB. The same is not true in code. You can easily build views and tell them to be exactly 360 points wide and 140 points wide at the same time. This can be mildly amusing if you're trying to make things fail, but it is more often utterly frustrating when you're trying to make things work, which is what most developers spend their time doing.

When rules fail, they fail loudly. Xcode provides you with verbose updates explaining what might have gone wrong. In some cases, your code will raise exceptions. Your app terminates if you haven't implemented handlers. In others (as in the example that follows), the Auto Layout keeps your app running by deleting conflicting constraint rules on your behalf. This produces interfaces that can be somewhat random.

Regardless of the situation, it's up to you to start debugging your code and your IB layouts to try to track down why things have broken and the source of the conflicting rules. This is not fun.

Consider the following console output. The text refers to that view I mentioned that attempts to be both 360 points and 140 points wide at the same time. The bolding in the text is mine. I've highlighted the sizes for each constraint, plus the reason for the error. In this example, both rules have the same priority and are inconsistent with each other:

```
2013-01-14 09:02:48.590 HelloWorld[69291:c07]
    Unable to simultaneously satisfy constraints.
Probably at least one of the constraints in the following list is one you
don't want. Try this: (1) look at each constraint and try to figure out which
you don't expect; (2) find the code that added the unwanted constraint or
constraints and fix it.
(Note: If you're seeing NSAutoresizingMaskLayoutConstraints that you don't
understand, refer to the documentation for the UIView property
translatesAutoresizingMaskIntoConstraints)
(
    "<NSLayoutConstraint:0x7147d40 H:[TestView:0x7147c50(360)]>",
    "<NSLayoutConstraint:0x7147e70 H:[TestView:0x7147c50(140)]>"
)

Will attempt to recover by breaking constraint
    <NSLayoutConstraint:0x7147d40 H:[TestView:0x7147c50(360)]>

Break on objc_exception_throw to catch this in the debugger.
The methods in the UIConstraintBasedLayoutDebugging category on
    UIView listed in <UIKit/UIView.h> may also be helpful.
```

This unsatisfiable conflict cannot be resolved except by breaking one of the constraints, which the Auto Layout system does. It arbitrarily discards one of the two size requests (in this case, the 360 size) and logs the results.

Sufficiency

Another key challenge is making sure that your rules are specific enough. An underconstrained interface (one that is *insufficient* or *ambiguous*) can create random results when faced with many possible layout solutions (see Figure 1-5, top). You might request that one view lies to the right of the other, but unless you tell the system otherwise, you might end up with the left view at the top of the screen and the right view at the bottom. That one rule doesn't say anything about vertical orientation.

Figure 1-5 Odd layout positions (top) are the hallmark of an underconstrained layout. Although these particular views are constrained to show up onscreen, their near-random layout indicates insufficient rules describing their positions. By default, views may not show up at all, especially when they are underconstrained. Chapter 4, “Visual Formats,” discusses fallback rules that ensure views are both visibly sized and onscreen. A sufficient layout (bottom) provides layout rules for each of its views.

A sufficient set of constraints fully expresses a view's layout, as in Figure 1-5 (bottom). In this image, each view has a well-defined size and position.

Sufficiency does not mean "hard coded." In this example, none of these positions are exactly specified. The Auto Layout rules say to place the views in a horizontal row, center-aligned vertically to each other. The first view is pinned off of the parent's left-center. These constraints are sufficient because every view's position can be determined from its relations to other views.

A sufficient or *unambiguous* layout generally offers at least two geometric rules per axis, or a minimum of four rules in all. For example, a view might have an origin and a size—as you would with frames—to specify where it is and how big it is. But, you can express much more with Auto Layout. The following sufficient rule examples define a view's position and extent along one axis, illustrated in Figure 1-6:

- You could pin the horizontal edges (A) of a view to exact positions in its superview. (The two properties defined in this example are the view's minimum X and maximum X positions.)
- You could match the width of one view to another subview (B), and then center it horizontally to its parent (width and center X).
- You could declare a view's width to match its intrinsic content, such as the length of text drawn on it (C), and then pin its right (trailing) edge to the left (leading) edge of another view (width and maximum X).
- You could pin the top and bottom of a view to the parent (D) so that the view stretches vertically along with its parent (minimum Y and maximum Y).
- You could specify a view's vertical center and its maximum extent (E), letting Auto Layout calculate the height from that offset (center Y and maximum Y).
- You could specify a view's height and its offset from the top of the view (F), hanging the view off the top of the parent (minimum Y and height.).

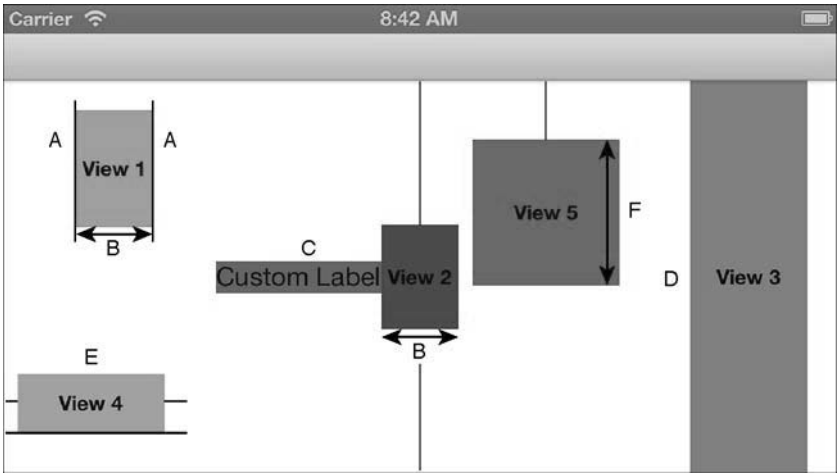

Figure 1-6 Examples of sufficient layout involve two rules per axis.

Each of these rules provides enough information along one axis to avoid ambiguity. That's because each one represents a specific declaration about how the view fits into the overall layout.

When rules fail, they lack this exactness. For example, if you supply only the width, where should the system place the item along the X-axis? At the left? Or the right? Somewhere in the middle? Or maybe entirely offscreen? Or if you only specify a Y position, how tall should the view be? 50 points? 50,000 points? 0 points? Missing information leads to ambiguous layouts.

You often encounter ambiguity when working with inequalities, as in the top image of Figure 1-5. The rules for these views say to stay within the bounds of the parent, but where? If their minimum X value is greater than or equal to their parent's minimum X value, what should that X value be? The rules are insufficient, and the layout is ambiguous.

Constraint Attributes

Constraints work with a limited geometric vocabulary. Attributes are the "nouns" of the constraint system, describing positions within a view's alignment rectangle. Relations are "verbs," specifying how the attributes compare to each other.

The attribute nouns (see Figure 1-7) speak to physical geometry. Constraints offer the following view attribute vocabulary:

- **left, right, top, and bottom**—The edges of a view's alignment rectangle on the left (A), right (B), top (C), and bottom (D) of the view. These correspond to a view's minimum X, maximum X, minimum Y, and maximum Y values.

- **leading and trailing**—The leading and trailing edges of the view's alignment rectangle. In left-to-right (English-like) systems, these correspond to "left" (leading, A)) and "right" (trailing, B). In right-to-left linguistic environments like Arabic or Hebrew, these roles flip; right is leading (B), and left is trailing (A).

Tip

When internationalizing your applications, always prefer leading and trailing over left and right. This allows your interfaces to flip properly when using right-to-left languages, like Arabic and Hebrew.

- **width and height**—The width (E) and height (F) of the view's alignment rectangle.
- **centerX and centerY**—The x-axis (H) and y-axis (G) centers of the views' alignment rectangle.
- **baseline**—The alignment rectangle's baseline (I), typically a set offset above its bottom attribute.

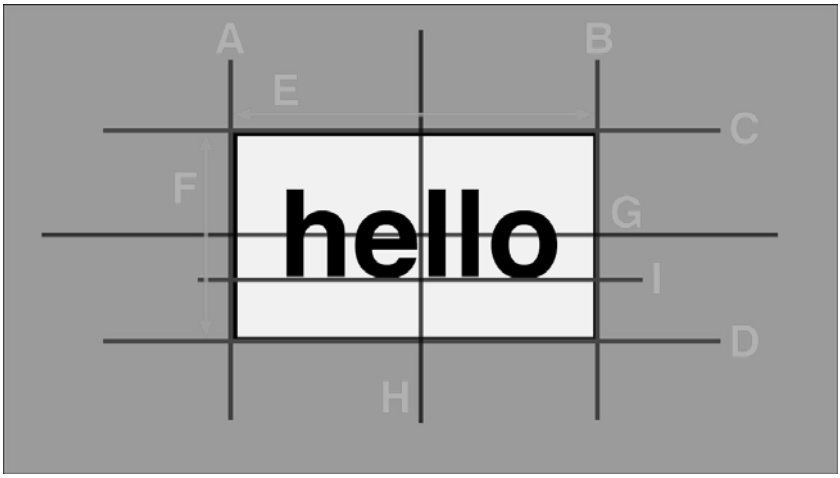

Figure 1-7 Attributes specify geometric elements of a view.

The relation verbs compare values. Constraint math is limited to three relations: setting equality or setting lower and upper bounds for comparison. You can use the following layout relations:

- **Less-than-or-equal inequality**—`NSLayoutRelationLessThanOrEqual`
- **Equality**—`NSLayoutRelationEqual`
- **Greater-than-or-equal inequality**—`NSLayoutRelationGreaterThanOrEqual`

This might not sound like a lot expressively. However, these three relations cover all the ground needed for specific equalities and for maximum and minimum limits.

Missing Views

It's common for developers new to Auto Layout to "lose" their views. They discover that views they have added end up offscreen or at that they have a zero size due to constraints. (Incidentally, Auto Layout works with positive sizes, zero or bigger. You cannot create views with negative widths and heights.) The missing views problem catches many devs. This problem happens both with underconstrained views and views with inconsistent rules.

In this section, you see a little bit of constraint code, even before you've read about the details of the constraint class and how instances work. Please bear with me. I've added highlights that help explain ambiguous and underconstrained scenarios to make a point. If you work with Auto Layout, you should be aware of these situations *before* you start using the technology.

Underconstrained Missing Views

With underconstrained views, there's not enough information for the Auto Layout system to build from, so it often defaults to a size of zero. Consider the following example. This code creates a new view, prepares it for Auto Layout, and then adds two sets of constraints, which I bolded:

```
// Create a new view and add it into the Auto Layout system
// This view goes missing despite the initWithFrame: size
UIView *view = [[UIView alloc]
    initWithFrame:CGRectMake(0.0f, 0.0f, 30.0f, 30.0f)];
[self.view addSubview:view];
view.translatesAutoresizingMaskIntoConstraints = NO;

// Add two sets of rules, pinning the view and setting height
[self.view addConstraints:[NSLayoutConstraint
    constraintsWithVisualFormat:@"V:|[view(==80)]" // 80 height
    options:0 metrics:nil
    views:NSDictionaryOfVariableBindings(view)]];
[self.view addConstraints:[NSLayoutConstraint
    constraintsWithVisualFormat:@"H:|[view]"
```

```
    options:0 metrics:nil
    views:NSDictionaryOfVariableBindings(view)]];
```

The first set of constraints pins the view to the top of its parent and sets the height to 80. The second set pins the view to the parent's leading edge. (This is the left side in the United States, with English's left-to-right writing system.) I deliberately did not specify a width. The view's size is, therefore, underconstrained.

You might expect Auto Layout to default to the initial frame size, which was set to 30 by 30 points. It does not. When this snippet set `translatesAutoresizingMaskIntoConstraints` to `NO`, that initialization was essentially thrown away. As the view appears onscreen, the ambiguous rules passed to Auto Layout result in a width that falls to zero, creating a nonvisible view:

```
2013-01-14 10:47:40.460 HelloWorld[73891:c07]
    <UIView: 0x884dfc0; frame = (0 0; 0 80); layer = <CALayer: 0x884e020>>
```

Missing Views with Inconsistent Rules

Inconsistent rules may also produce views that are missing in action. For example, imagine a pair of rules that say "View A is three times the width of View B" and "View B is twice the width of View A." The following code snippets implement these rules. I've highlighted the parts of the code that tell the rule story:

```
NSLayoutConstraint *constraint;
constraint = [NSLayoutConstraint
    constraintWithItem:viewA
    attribute:NSLayoutAttributeWidth
    relatedBy:NSLayoutRelationEqual
    toItem:viewB
    attribute:NSLayoutAttributeWidth
    multiplier:3.0f constant:0.0f];
[self.view addConstraint:constraint];

constraint = [NSLayoutConstraint
    constraintWithItem:viewA
    attribute:NSLayoutAttributeWidth
    relatedBy:NSLayoutRelationEqual
    toItem:viewB
    attribute:NSLayoutAttributeWidth
    multiplier:2.0f constant:0.0f];
[self.view addConstraint:constraint];
```

Surprisingly, these two rules are neither unsatisfiable nor ambiguous, even though common sense suggests otherwise. That's because both rules are satisfied when View A and View B have zero width. At a zero size, View A's width can be three times the width of View B, and View B twice the width of View A:

```
0 = 0 * 3 and 0 = 0 * 2
```

When this code is run and the rules applied, the views present the zero-width frames expected from this scenario:

```
2013-01-14 11:02:38.005 HelloWorld[74460:c07]
    <TestView: 0x8b30910; frame = (320 454; 0 50); layer = <CALayer: 0x8b309d0>>
2013-01-14 11:02:38.006 HelloWorld[74460:c07]
    <TestView: 0x8b32570; frame = (320 436; 0 68); layer = <CALayer: 0x8b32450>>
```

Tracking Missing Views

You can track down "missing" views with the debugger by inspecting their geometry after you expect them to appear (for example, `viewWillAppear:` and `awakeFromNib`). You may want to add `NSAssert` statements about their expected size and positions. Some will be, as discussed, zero sized.

The following view, for example, had a zero-sized frame because it was underconstrained in the Auto Layout system:

```
2013-01-09 14:31:41.869 HelloWorld[29921:c07] View: <UIView: 0x71bb390;
frame = (30 430; 0 0); layer = <CALayer: 0x71bb3f0>>
```

Others may simply be offscreen because you haven't told Auto Layout that the views must appear onscreen. This view had a positive size (20 points by 20 points), but its frame with its (–20, –20) origin lay outside of its view controller's presentation:

```
2013-01-09 14:33:37.546 HelloWorld[29975:c07] View: <UIView: 0x7125f70;
frame = (-20 -20; 20 20); layer = <CALayer: 0x7125fd0>>
```

In other cases, you might load a view from a storyboard or NIB and see only part of it onscreen, or it may occupy the entire screen at once. These are hallmarks of an underlying Auto Layout issue.

Ambiguous Layout

During development, you can test whether a view's constraints are sufficient by calling `hasAmbiguousLayout`. This returns a Boolean value of `YES` for a view that could have occupied a different frame and `NO` for a view whose constraints are fully specified.

These results are view specific. For example, imagine a fully constrained view whose child is underconstrained. The view itself does not have ambiguous layout, even though its child does. You can and should test the layout individually for each view in your hierarchy, as follows:

```
@implementation VIEW_CLASS (AmbiguityTests)
// Debug only. Do not ship with this code
- (void) testAmbiguity
{
    NSLog(@"<%@:0x%0x>: %@",
```

```
        self.class.description, (int)self,
        self.hasAmbiguousLayout ? @"Ambiguous" : @"Unambiguous");

    for (VIEW_CLASS *view in self.subviews)
        [view testAmbiguity];
}
@end
```

> **Note**
>
> In this code snippet, and throughout this book, `VIEW_CLASS` is defined as either `UIView` or `NSView`, depending on the system of deployment.

This code descends through a view hierarchy and lists the results for each level. Here's what a simple layout with two subviews returned for the underconstrained layout code originally shown in Figure 1-5 (top). The parent view does not express ambiguous layout but its child views do:

```
HelloWorld[76351:c07] <UIView:0x715a9a0>: Unambiguous
HelloWorld[76351:c07] <TestView:0x715add0>: Ambiguous
HelloWorld[76351:c07] <TestView:0x715c9e0>: Ambiguous
```

You can run these tests as soon as you like—in `loadView` or wherever you set up new views and add constraints. It's generally a good first step for any time you're adding new views to your system as well. It ensures that your constraints really are as fully specified as you *think* they are.

You do not need to test any views produced in Interface Builder. IB guarantees that the rules it produces are unambiguous.

Exercising Ambiguity

Apple offers a curious tool in the form of its `exerciseAmbiguityInLayout` view method. It automatically tweaks view frames that express ambiguous layouts. This is a view method (`UIView` and `NSView`) that checks for ambiguous layout and attempts to randomly change a view's frame.

Figure 1-8 shows this call in action. Here, you see an OS X window with three underconstrained subviews. These views appear in gray (bottom left), green (top right), and tan (bottom right). Their positions have not been set programmatically, so they end up wherever Auto Layout places them. In this example, after exercising ambiguity (see Figure 1-8, right), the tan view moves to the bottom left.

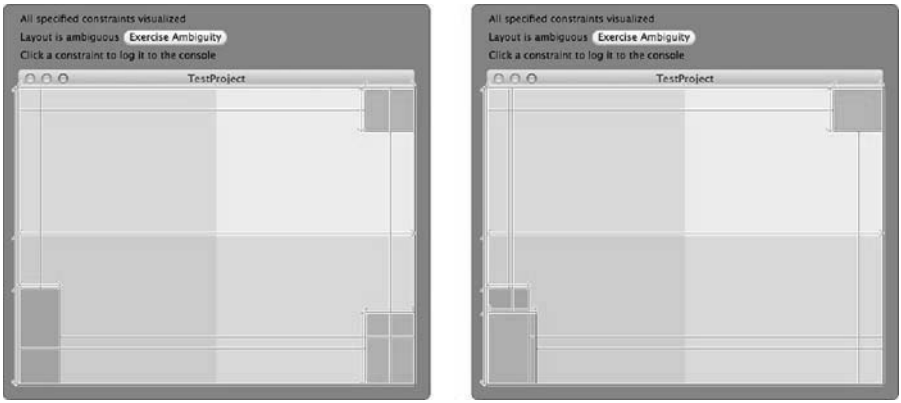

Figure 1-8 Exercising ambiguity allows you to change view frames to other legal values allowable under your current set of Auto Layout constraints.

This tells you that (1) this is one of the affected underconstrained views and (2) you can see some of the range that might apply to this view due to its lack of positioning constraints.

Exercising ambiguity is a blunt and limited weapon. In this example, the green and gray views are unchanged, even though they also had ambiguous layout. Don't relay exercising ambiguity to exhaustively find issues in your project, although it can be a useful tool for the right audience.

I am not a big fan of this feature, but, to be absolutely honest, it *has* helped me out of a pickle or two.

Visualizing Constraints

A purple outline surrounds the window in Figure 1-8. This is an OS X-only feature. On OS X, you can visualize constraints by calling `visualizeConstraints:` on any `NSWindow` instance. You pass it an array of constraints you want to view.

Here is a simple way to exhaustively grab the constraints from a view and all its subviews using simple class extension:

```
@implementation VIEW_CLASS (GeneralConstraintSupport)
// Return all constraints from self and subviews
- (NSArray *) allConstraints
{
    NSMutableArray *array = [NSMutableArray array];
    [array addObjectsFromArray:self.constraints];
    for (VIEW_CLASS *view in self.subviews)
        [array addObjectsFromArray:[view allConstraints]];
```

```
    return array;
}
@end
```

> **Note**
>
> Apple can and does regularly extend classes. When creating categories for production code, do *not* use obvious names (like `allConstraints`) that may conflict with Apple's own development. Adding custom prefixes, typically company initials, guards your code against conflicts with potential future updates. This book does not follow its own advice solely to make the code more readable.

The purple backdrop that appears tells you whether the window's layout is ambiguous. It tests from the window down its view hierarchy all the way to its leaves. If it finds any ambiguity, it offers the Exercise Ambiguity button, which means that you don't have to call the option from your own code.

This option also shows you the constraints you passed as clickable blue lines, helping you visualize those constraints in a live application. Click any item to log it to the Xcode debugging console.

All of these methods—testing for ambiguous layout, exercising that layout ambiguity, and visualizing constraints—are meant for development builds only. Don't ship production code that calls them.

Intrinsic Content Size

Under Auto Layout, a view's content plays as important a role in its layout as its constraints. This is expressed through each view's `intrinsicContentSize`. This size describes the minimum space needed to express the full view content without squeezing or clipping that data. It derives from the natural properties of the content each view presents.

For an image view, for example, this corresponds to the size of the image it presents. A larger image requires a larger intrinsic content size. Consider the following code snippet. It loads a standard Icon.png image into an image view and reports the view's intrinsic content size. As you'd expect, this size is 57 by 57 points, the size of the image supplied to the view (see Figure 1-9, top):

```
UIImageView *iv = [[UIImageView alloc]
    initWithImage:[UIImage imageNamed:@"Icon.png"]];
NSLog(@"%@", NSStringFromCGSize(iv.intrinsicContentSize));
```

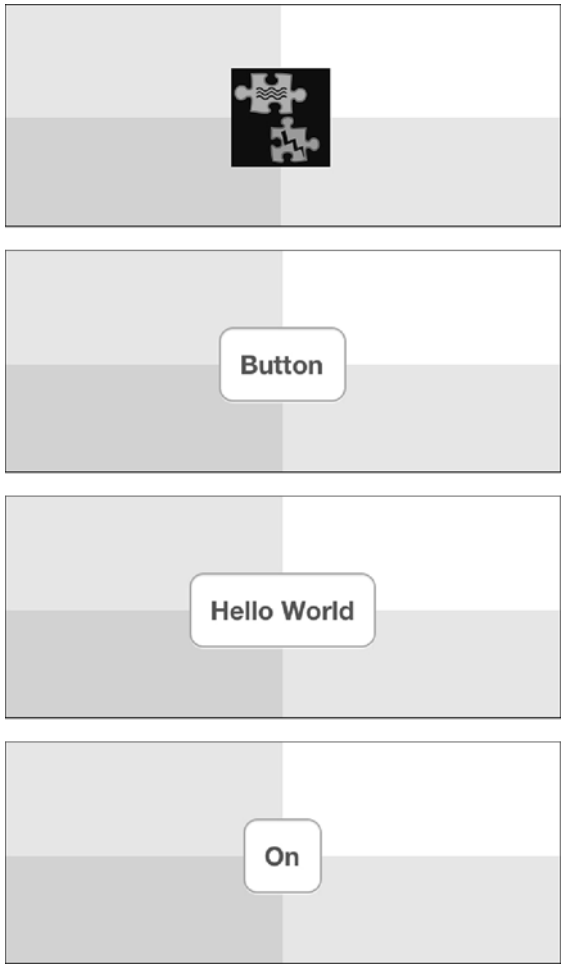

Figure 1-9 A view's intrinsic content size relates the natural size that its contents occupy.

For a button, this varies with its title (see the three button images in Figure 1-9). As a title grows or shrinks, the button's intrinsic content size adjusts to match. This snippet creates a button and assigns it a pair of titles, reporting the intrinsic content size after each assignment:

```
UIButton *button =
    [UIButton buttonWithType:UIButtonTypeRoundedRect];

// Longer title, Figure 1-9, bottom-left image
[button setTitle:@"Hello World" forState:UIControlStateNormal];
NSLog(@"%@: %@", [button titleForState:UIControlStateNormal],
```

```
    NSStringFromCGSize(button.intrinsicContentSize));

// Shorter title, Figure 1-9, bottom-right image
[button setTitle:@"On" forState:UIControlStateNormal];
NSLog(@"%@: %@", [button titleForState:UIControlStateNormal],
    NSStringFromCGSize(button.intrinsicContentSize));
```

When run, this snippet outputs the following sizes. The Hello World version of the button expresses a wider intrinsic content size than the On version, and both use the same height. These values can vary further as you customize a font face and font size and title text:

```
2013-01-11 12:16:46.616 HelloWorld[47516:c07] Hello World: {107, 43}
2013-01-11 12:16:46.616 HelloWorld[47516:c07] On: {45, 43}
```

A view's intrinsic size allows Auto Layout to best match a view's frame to its natural content. Earlier, you read that unambiguous layout generally requires setting two attributes in each axis. When a view has an intrinsic content size, that size accounts for one of the two attributes. You can, for example, place a text-based control or an image view in the center of its parent and its layout will not be ambiguous. The intrinsic content size plus the location combine for a fully specified placement.

When you change a view's intrinsic contents, call `invalidateIntrinsicContentSize` to let Auto Layout know to recalculate at its next layout pass.

Compression Resistance and Content Hugging

As the name suggests, *compression resistance* refers to the way a view protects its content. A view with a high compression resistance fights against shrinking. It won't allow that content to clip. Consider the buttons at the right of Figure 1-10. Both screenshots show an application responding to a constraint that wants to set that button width to 40 points.

Figure 1-10 Compression resistance describes how a view attempts to maintain its minimum intrinsic content size. The top screenshot's button has a high compression resistance.

The top version uses a high compression resistance value and the bottom version uses a lower value. As you can see, the higher priority ensures that the top button succeeds in preserving its intrinsic content. In the case of the bottom button, its resistance is too low. The resizing succeeds and the button compresses, clipping the text.

The button's "don't clip" request is still there, but it's no longer important enough to prevent the "please set the width to 40" constraint from resizing the view to the button's detriment. Auto Layout often comes across two conflicting requests. When only one of those requests can win, it satisfies the one with the higher priority.

You specify a view's compression resistance through Interface Builder's Size Inspector (View > Utilities > Show Size Inspector, View > Content Compression Resistance Priority), as shown in Figure 1-11, or by setting a value in code. Set the value separately for each axis, horizontal and vertical. Values may range from 1 (lowest priority) to 1,000 (required priority):

```
[button setContentCompressionResistancePriority:500
    forAxis:UILayoutConstraintAxisHorizontal];
```

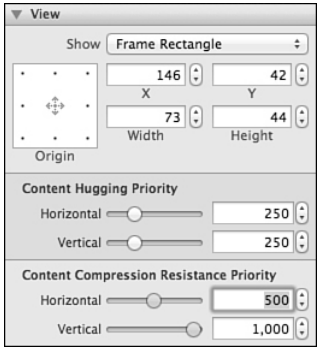

Figure 1-11 Adjust a view's Content Compression Resistance and Content Hugging priorities in IB's Size Inspector or through code. Although these numbers are presented as a scale of positive integers in IB, they're actually typed as floats: `typedef float UILayoutPriority` (iOS) and `NSLayoutPriority` (OS X).

This inspector is also where you set a view's *content hugging* priority. This refers to the way a view prefers to avoid extra padding around its core content (as shown here) or stretching of that core content (as with an image view that uses a scaled content mode). The buttons in Figure 1-12 are being told to stretch to meet up with label at the left.

The button at the top has a high content hugging priority, so it resists that stretching. It hugs to the content (in this case, the word *Button*). The button in the bottom screenshot has a lower content hugging priority. The request to stretch left wins out. The button pads its contents and produces the wide result you see.

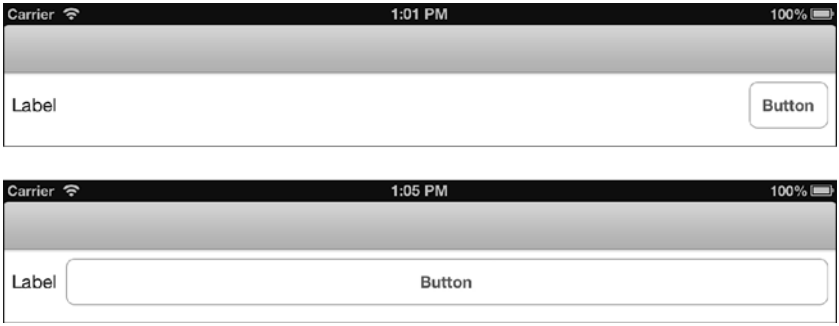

Figure 1-12 Content hugging describes a view's desire to match its frame to the natural size of its content. A strong hugging priority limits the view from growing much larger than the content it presents. A weak priority may allow a view to stretch and isolate its content among a sea of padding.

As with compression resistance, you set a view's hugging priority in IB's Size Inspector (refer to Figure 1-11) or in code:

```
[button setContentHuggingPriority:501
    forAxis:UILayoutConstraintAxisHorizontal]
```

Auto Layout and Frames

With Auto Layout, view frames play a new, less important role. In an Auto Layout world, frames are now the guys in an opera who stands around holding a spear while the constraints and intrinsic content sizes sing their arias. The focus moves from specific sizes and placement to suggestions and relations.

When you worked with Autosizing, frames said where to place views on the screen and how big those views would be. In Auto Layout, constraints determine view size and placement using a geometric element called an *alignment rectangle.*

As developers create complex views, they may introduce visual ornamentation such as shadows, exterior highlights, reflections, and engraving lines. As they do, these features usually become attached as subviews or sublayers or integrated into a view's image. As a consequence, a view's full extent may grow as items are added.

Unlike frames, a view's alignment rectangle is limited to a core visual element. Its size remains unaffected as new items join the primary view. Consider Figure 1-13 (left). It depicts a view with an attached shadow and a badge. When laying out this view, you want Auto Layout to focus on aligning just the core element.

Figure 1-13 A view's alignment rectangle (center) refers strictly to the core visual element to be aligned, without embellishments.

The center image shows the view's alignment rectangle. This rectangle excludes all ornamentation, including the drop shadow and badge. It's the part of the view you want considered when Auto Layout does its work. Contrast this with the rectangle shown in the right image of Figure 1-13. This version includes all the visual ornamentation, extending the view's frame beyond the area that should be considered for alignment.

This rectangle encompasses all the view's visual elements. It includes the shadow and badge. These ornaments could potentially throw off a view's alignment features (for example, its center, bottom, and right) if they were considered during layout.

By working with alignment rectangles instead of frames, Auto Layout ensures that key information, like a view's edges and center, are properly considered during layout. In Figure 1-14, the adorned view is perfectly aligned on the background grid. Its badge and shadow are not considered during placement.

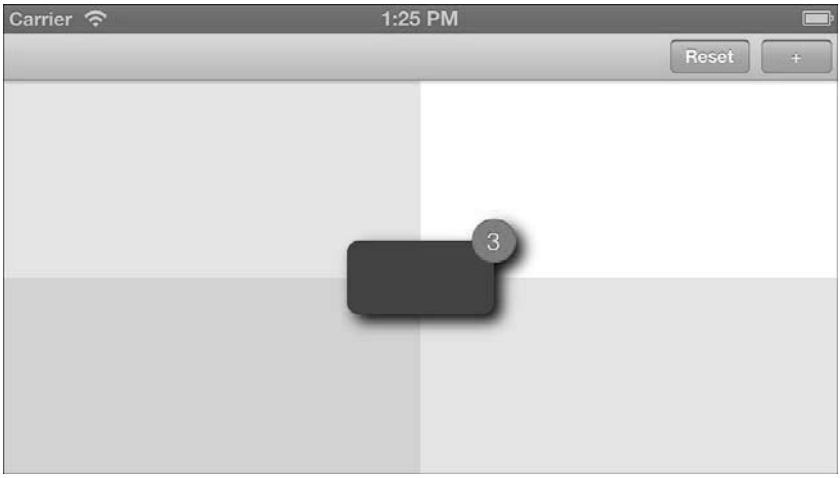

Figure 1-14 Auto Layout only considers this view's alignment rectangle when laying it out as centered in its parent. The shadow and badge don't affect its placement.

Visualizing Alignment Rectangles

Both iOS and OS X enable you to overlay views with their alignment rectangles in your running application. You set a simple launch argument from your app's scheme. This is `UIViewShowAlignmentRects` for iOS and `NSViewShowAlignmentRects` for OS X. Set the argument value to `YES` and make sure to prefix with a dash, as shown in Figure 1-15.

When the app runs, rectangles show over each view. The resulting rectangles are light and can be difficult to see.

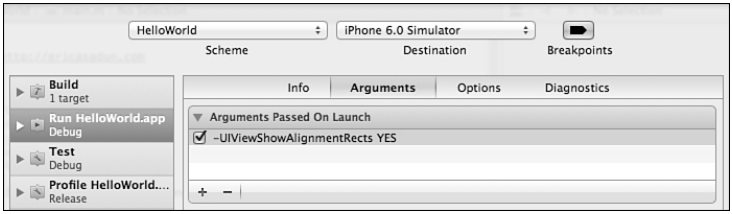

Figure 1-15 Set launch arguments in the scheme editor.

Image Alignment Insets

Alignment issues affect the way you handle images. Image art often contains hard-coded embellishments, like highlights, shadows, and so forth. These items take up little memory and run more efficiently than those created by adding layer effects. Because of that, many developers use images enhancements in preference to Quartz 2D effects because of their low overhead.

Figure 1-16 demonstrates the typical problem encountered when using image-based ornamentation with Auto Layout. The left image shows a basic image view, whose art I created in Photoshop. I used a standard drop shadow effect. When added to the image view, the 20-point by 20-point area I left for the shadow throws off the view's alignment rectangle, causing it to appear slightly too high and left.

In its default implementation, the image view has no idea that the image contains ornamental elements. You have to tell it how to adjust its intrinsic content so that the alignment rectangle considers just that core material.

To accommodate the shadow, you load and then rebuild the image. This is a two-step process. First, you load the image as you normally would (for example, with `imageNamed:`). Then, you call `imageWithAlignmentRectInsets:` on that image to produce a new version supporting the specified insets:

```
UIImage *image = [[UIImage imageNamed:@"Shadowed.png"]
    imageWithAlignmentRectInsets:UIEdgeInsetsMake(0, 0, 20, 20)];
UIImageView *imageView = [[UIImageView alloc] initWithImage:image];
```

After specifying the alignment rect insets, the updated version now properly aligns, as you see in Figure 1-16, right. I logged out the pertinent details so that you can compare the view details. Here's what the view frame looks like (it shows the full 200×200 image size), the intrinsic content size built from the image's alignment insets (180×180), and the resulting alignment rectangle used to center the image view's frame:

```
HelloWorld[53122:c07] Frame: {{70, 162}, {200, 200}}
HelloWorld[53122:c07] Intrinsic Content Size: {180, 180}
HelloWorld[53122:c07] Alignment Rect: {{70, 162}, {180, 180}}
```

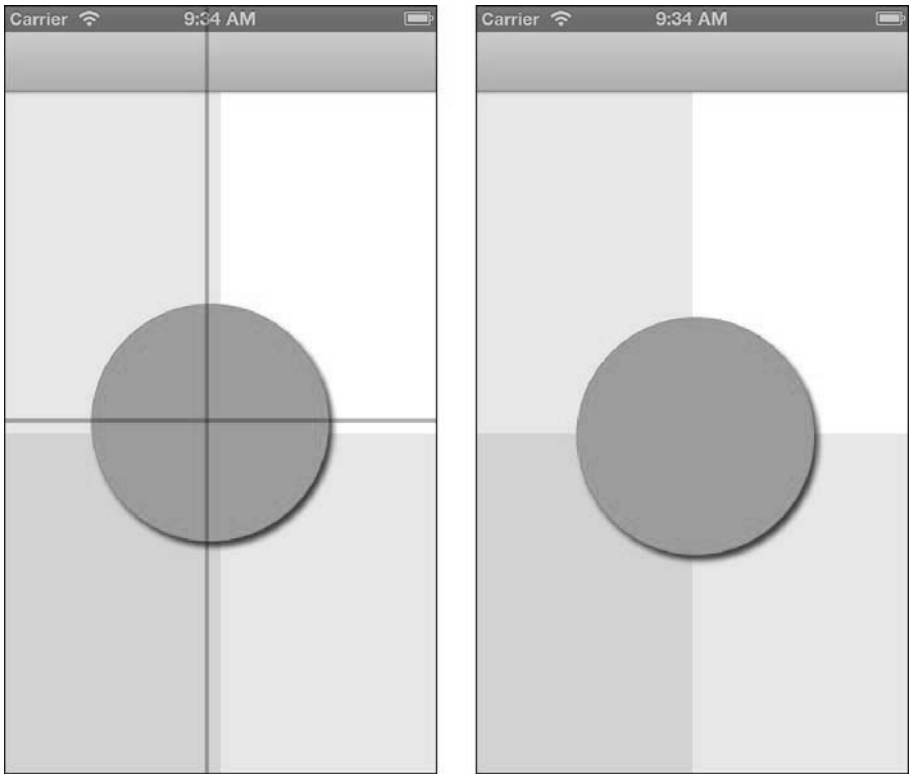

Figure 1-16 Adjust your images to account for alignment when using Auto Layout. At the left, the image view was created with a raw, unadjusted image. It displays slightly too far left and up, which you can inspect by looking at the points where the circle crosses the background grid. I added lines over the screenshot to emphasize where the centering should have occurred. The right screenshot shows the adjusted image view. It centers exactly onto its parent view.

Declaring Alignment Rectangles

Cocoa and Cocoa Touch offer a several additional ways to report alignment geometry. You may implement `alignmentRectForFrame:`, `frameForAlignmentRect:`, `baselineOffsetFromBottom`, and `alignmentRectInsets`. These methods allow your views to declare and translate alignment rectangles from code.

For the most part, thankfully, you can ignore alignment rectangles and insets. Things just, for the most part, work. The edge cases you encounter usually happen when Auto Layout comes into conflict with transforms (and other circumstances when the actual frame doesn't match the visual frame as with buttons).

A few notes on these items:

- `alignmentRectForFrame:` and `frameForAlignmentRect:` must always be mathematical inverses of each other.
- Most custom views only need to override `alignmentRectInsets` to report content location within their frame.
- `baselineOffsetFromBottom` is available only for `NSView` and refers to the distance between the bottom of a view's alignment rectangle and the view's content baseline, such as that used for laying out text. This is important when you want to align views to text baselines and not to the lowest point reached by typography descenders, like *j* and *q*.

Here's some information about `alignmentRectForFrame:` and `frameForAlignmentRect:` from the UIView.h documentation:

> These two methods should be inverses of each other. UIKit will call both as part of layout computation. They may be overridden to provide arbitrary transforms between frame and alignment rect, though the two methods must be inverses of each other. However, the default implementation uses alignmentRectInsets, so just override that if it's applicable. It's easier to get right.
>
> A view that displayed an image with some ornament would typically override these, because the ornamental part of an image would scale up with the size of the frame. Set the NSUserDefault UIViewShowAlignmentRects to YES to see alignment rects drawn.

NSLayoutConstraint.h on OS X adds the following comment:

> If you do override these be sure to account for the top of your frame being either minY or maxY depending on the superview's flippedness.

You'll see this flippedness adjustment made in Listing 1-1, which is introduced in the next section.

Implementing Alignment Rectangles

Listing 1-1 offers a trivial example of code-based alignment geometry. This OS X app builds a fixed-size view and draws a shadowed rounded rectangle into it. When `USE_ALIGNMENT_RECTS` is set to 1, its `alignmentRectForFrame:` and `frameForAlignmentRect:` methods convert to and from frames and alignment rects. As Figure 1-17 shows, these reporting methods allow the view to display with proper alignment.

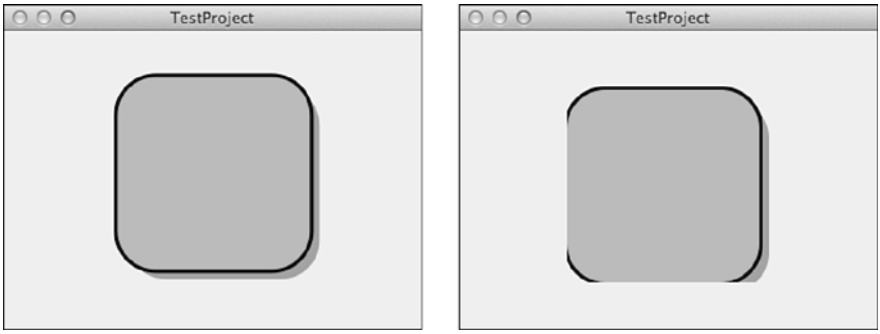

Figure 1-17 Implementing intrinsic content size and frame/alignment rect conversion methods ensures that your view will align and display correctly (left) rather than be misaligned and clipped (right).

Listing 1-1 **Using Code-Based Alignment Frames**

```
@interface CustomView : NSView
@end

@implementation CustomView
- (void) drawRect:(NSRect)dirtyRect
{
    NSBezierPath *path;

    // Calculate offset from frame for 170x170 art
    CGFloat dx = (self.frame.size.width - 170) / 2.0f;
    CGFloat dy = (self.frame.size.height - 170);

    // Draw a shadow
    NSRect rect = NSMakeRect(8 + dx, -8 + dy, 160, 160);
    path = [NSBezierPath
        bezierPathWithRoundedRect:rect xRadius:32 yRadius:32];
    [[[NSColor blackColor] colorWithAlphaComponent:0.3f] set];
    [path fill];
```

```
    // Draw fixed-size shape with outline
    rect.origin = CGPointMake(dx, dy);
    path = [NSBezierPath
        bezierPathWithRoundedRect:rect xRadius:32 yRadius:32];
    [[NSColor blackColor] set];
    path.lineWidth = 6;
    [path stroke];
    [ORANGE_COLOR set];
    [path fill];
}

- (NSSize)intrinsicContentSize
{
    // Fixed content size - base + frame
    return NSMakeSize(170, 170);
}

#define USE_ALIGNMENT_RECTS 1
#if USE_ALIGNMENT_RECTS
- (NSRect)frameForAlignmentRect:(NSRect)alignmentRect
{
    // 1 + 10 / 160 = 1.0625
    NSRect rect = (NSRect){.origin = alignmentRect.origin};
    rect.size.width = alignmentRect.size.width * 1.06250;
    rect.size.height = alignmentRect.size.height * 1.06250;
    return rect;
}

- (NSRect)alignmentRectForFrame:(NSRect)frame
{
    // 160 / 170 = 0.94117
    // Account for vertical flippage
    CGFloat dy = (frame.size.height - 170) / 2.0;
    rect.origin = CGPointMake(frame.origin.x, frame.origin.y + dy);

    rect.size.width = frame.size.width * 0.94117;
    rect.size.height = frame.size.height * 0.94117;
    return rect;
}
#endif
@end
```

Summary

This chapter introduced the core concepts that underpin Auto Layout, Cocoa's declarative constraint-based descriptive layout system. You learned that Auto Layout focuses on the relationships between views, and between views and their content, instead of on their frames. A logical priority-based framework drives Auto Layout. You discovered that its rules must be satisfiable, consistent, and sufficient. Here are a few final thoughts to take away with you from this chapter:

- Constraints are fun and powerful, and they provide elegant solutions to common layout situations. Do not be put off by Interface Builder's layout editor, which can be disappointing for nontrivial layout; constraints are brilliant!
- Don't be afraid to mix and match Auto Layout and Autosizing. So long as their rules do not conflict, you may port your existing layouts to a new world.
- Auto Layout is more than just constraints. Its content-protecting features provide a key component that helps specify what to show and not just where to show it. Compression resistance and content hugging play major roles in adapting graphical user interfaces (GUIs) during internationalization. When languages change, labels can vary widely in size.
- Several projects are underway attempting to backport Auto Layout to iOS 5 and earlier. Search around the Web for details. None of these projects have made significant headway yet, but they're worth keeping an eye on.