

MOBILE
PROGRAMMING
SERIES



iOS

INTERNATIONALIZATION

The Complete Guide

SHAWN E. LARSON

FREE SAMPLE CHAPTER

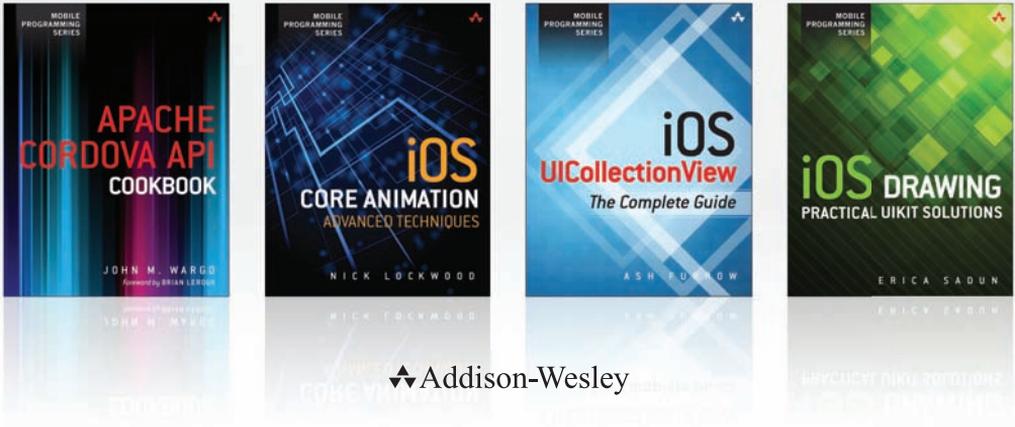
SHARE WITH OTHERS



iOS

Internationalization:
The Complete
Guide

Addison-Wesley Mobile Programming Series



Visit informit.com/mobile for a complete list of available publications.

The **Addison-Wesley Mobile Programming Series** is a collection of programming guides that explore key mobile programming features and topics in-depth. The sample code in each title is downloadable and can be used in your own projects. Each topic is covered in as much detail as possible with plenty of visual examples, tips, and step-by-step instructions. When you complete one of these titles, you'll have all the information and code you will need to build that feature into your own mobile application.



Make sure to connect with us!
informit.com/socialconnect

informIT.com
the trusted technology learning source

 Addison-Wesley

Safari
Books Online

iOS

Internationalization:

The Complete

Guide

Shawn E. Larson

◆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

iOS Internationalization: The Complete Guide

Copyright © 2015 by Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-134-03772-1

ISBN-10: 0-134-03772-3

09 08 07 06 4 3 2 1

iOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

App Store, Apple, the Apple logo, Cocoa, Finder, Interface Builder, iPad, iPhone, iPod, iPod touch, iTunes, Mac, Macintosh, Objective-C, Safari, and Xcode are trademarks of Apple, Inc., registered in the United States and other countries.

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.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Editor-in-Chief

Mark Taub

Senior Acquisitions Editor

Trina MacDonald

Development Editor

Sheri Replin

Managing Editor

Kristy Hart

Project Editor

Elaine Wiley

Copy Editor

Cheri Clark

Proofreader

Language Logistics,
Chrissy White

Technical Reviewer

Nick Lockwood

Editorial Assistant

Olivia Basegio

Cover Designer

Chuti Prasertsith

Layout

Cook



To Amy,

My friend. My love. My constant. My gift.



Contents at a Glance

Introduction

1 International Settings

2 Characters and Encoding

3 Coding for Locale

4 Prepping Your App for Localization

5 Localizing Your App

6 Adjusting the UI

7 Submitting Your App

A Case Study: *Boom Beach*

B Web Resources

Table of Contents

Introduction

- Reader Expectations
- How This Book Is Organized
- Getting the Sample Code
- Contacting the Author

1 International Settings

- System Settings: International
 - Language
 - Voice Control
 - Siri
 - Voice Dial Only
 - Keyboards
 - Calendars
 - Gregorian
 - Japanese
 - Buddhist
 - Region Format
 - Representing Dates
 - Long Versus Short Date Formats
 - Representing Time
 - Telephone
 - Currency
 - Sorting
 - Measurements System
 - Decimal and Thousands Separators
 - Quotation Marks
 - Other Callouts
 - Vertical Text
 - List Separator
 - Summary

2 Characters and Encoding

- Characters

 - Types of Characters

 - Accented Characters

 - Chinese Characters

- Strings

- Code Pages and Encoding

 - ASCII Character Set

 - Extended Character Set

 - ANSI Standard

 - Asian Character Support: DBCS

- Unicode and Encoding

 - Unicode Planes

 - Combining Character Sequences

 - Duplicate Characters

- Encoding in Action

 - UTF-8

 - UTF-16

- Coding Encoding

 - Example 1: Encoding to ASCII

 - Example 2: Returning the ASCII Value from a Character

 - Example 3: Encoding an ASCII String to UTF-8

 - Example 4: Returning a String from an Encoding URL

- Objective-C Encoding Enumerations

- Encoding Gone Bad

- Diacritics

 - Precomposed Diacritics

- Surrogate Characters

 - Emoji

- Retrieving Characters from Unicode Code Points

- Obtaining Unicode Code Points

- Glyphs

 - Contextual Glyphs

 - Fonts

Ligatures

Code Snippet to Compare Ligatures

Summary

3 Coding for Locale

`NSLocale` Class

Locale ID

`NSLocale` Component Keys

Return Types from `NSLocale`'s Component Keys

Auto Updating the Locale

Address Book Framework

`NSNumberFormatter`

`NSDate`

`NSDateFormatter`

Predefined Styles

Format Specifiers

Conversion

Date Templates

Date Format "j" Template

`NSCalendar`

`NSDateComponents`

Components from a Date

Relative Date Calculations

`NSTimeZone`

`NSString`

Initializing a String

Lower- and Uppercase

Searching

Folding Strings

Sorting

Summary

4 Prepping Your App for Localization

Building an International Exerciser

App Creation Overview

Covered Locale Categories

View Controller Creation

Locale Components Affected by Setting Changes

Auto Updating Locale

Home Screen

Table View Controller

Measurement System

Quotes and Scripts

Calendar and Date Components

Date Formatting

Short, Medium, Long, Full Date Styles

Date Strings to Date Objects

Number Formatting

Decimal, Currency, Percent, and Spelled Out

Contact Names/Address Book Framework

Strings

String Case

Sorting

Exemplar Character Set

Other Locale-Specific Topics

Right-to-Left Languages

Telephone Format

Summary

5 Localizing Your App

Localization

What Is Localization?

What Gets Localized?

How Do the “Dot-Strings” Files Get Created?

Where Do the Strings Get Parsed?

Generalized Steps to Localize

Localizing a View Controller from the Sample App

Adding Localization to a Project

Localizing Code

The `genstrings` Command

Copying `Localizable.strings` Files

Xcode Reference

Localizing UI

Running Under the Localized Language

Translation Results with Japanese

Working with Localized Images

Localizing App Icon Name

Other NSLocalizedString Macros

`NSLocalizedStringFromTable()`

`NSLocalizedStringFromTableInBundle()`

`NSLocalizedStringWithDefaultValue()`

Creating a Generalized Key Name

genstrings Tools

`Update_Localization`

`pygenstrings`

Translating to Localized Languages

No “Mad Libs”

Avoid Translation Templates

Translation Services

Importing Files from Translators

Word Order

Plural Forms and the CLDR

What Do We Mean by Quantities?

Plurals Example

Genders

Summary

6 Adjusting the UI

Auto Layout and Constraints

Internationalization Implications

Avoiding Clipping

Right-to-Left Languages

Keyboards and Screen Coverage

Tools and Testing

Pseudo Localization

Launch Switches

Previewing Rotation in Xcode

UI Localization

Images

Right-to-Left Images

- Colors
- String Length
- Single-Letter Icons
- Body-Part Metaphors
- Locale Maps
- Summary

7 Submitting Your App

- App Submission Requirements
 - iOS Dev Center
 - Provisioning Profiles
 - iTunes Connect
- App Store Details
 - Available Territories
 - Pricing Tiers
- Localizing iTunes Connect Data
 - Changing Available Territories
 - Language Mapping
 - Adding Language Support
 - Field Breakout
- Apple Promotional Materials
- Summary

A Case Study: *Boom Beach*

B Web Resources

About the Author

Shawn E. Larson is a graduate of the University of Idaho and has more than 20 years of experience in the software industry. As a member of the Microsoft Office engineering team, he contributed to Mac Office and Office for the iPad development. His day jobs now are as an iOS dev for Nordstrom and an instructor for the iOS certification program at the University of Washington. In his off time, Shawn enjoys all things geeky and gadgety, the outdoors, and his wife's Zumba class. Shawn and Amy have been married for more than 20 years and have two children, Lindsey and Michael.

Acknowledgments

I had a random selection of folks who helped me tackle topics in writing this book. If I've forgotten anyone, it was unintentional!

Thanks to Rich Schaut, for my first interview for this book and getting this book project started.

Eric Paquin, thank you for being so responsive to my questions, reviewing my proposal, and giving suggestions.

Kyle Sluder, thanks for taking the time to go over your `TextViewWritingDirection` GitHub project.

Much appreciation to my translation buddies: to Joachim Hill-Grannec for the French translations and to Yuji Hayano for his gracious, never-ending help in translating and supplying Japanese strings (ありがとう).

Can Berk Güder, thanks for helping me complete the “Turkey test”—getting me real Turkish words using the Turkish “i.”

Jeremiah Johnson and Cody Vandermyn, thanks for letting me peek over your shoulders to get a better handle on working with iTunes Connect. Thanks also for the answers to the various and sundry iOS development questions you've fielded frequently.

Many thanks to my manager at Nordstrom, Keith Willsey, who encouraged our team to participate in the Seattle iOS Developer Meetups, which led to my extension lecture position at the University of Washington, which led to getting contacted by Trina MacDonald, which led to this book coming into existence.

Tim Ekl, thanks for your patience in working through my right-to-left language implications of Auto Layout and constraints at our “localized lunches.”

Jason Christensen, thanks for being “on call” with your Unicode help!

Thanks to Trina MacDonald for presenting this opportunity and support and to Olivia Basegio for keeping things rolling steady behind the scenes.

I've accumulated technical debt to Nick Lockwood for his tech review of my chapters, keeping me honest, and not missing the things I missed.

Thanks to friends, family, and colleagues for their encouragement and keeping me going by always asking, “How's the book going?”

And my greatest appreciation goes to my wife, Amy, for first being the “teaching widow” and then the “book widow.” Your support and belief in me means the world to me. Thank you so much, love! I'm almost done; I'll be back for our weekends and evenings!

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 email or write me directly to let us 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 we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we 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 email address.

Email: trina.macdonald@pearson.com

Mail: Reader Feedback
Addison-Wesley Mobile Programming
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

Welcome! Accueil! Przywitać! 歓迎

Welcome to your first taste of localization: the word “welcome” translated into the four languages English, French, Polish, and Japanese. But this book is about more than localization and translating strings. This book covers *internationalization*. What is internationalization, and how does that differ from localization? Although these terms do tend to become interchangeable, their applications are not equivocal.

Internationalization is the process of preparing your app to be localized. It encompasses supporting the cultural information of a given region and locale. It is preparing your code to support the character set, calendars, number format, sorting, and text direction of that locale, just to name a few. You often see the term “internationalization” abbreviated with the numeronym “i18n.”

Localization is the process of translating text and assets to support a specified region’s language. It’s often abbreviated “l10n.”

So that’s the “what”; what about the “why?” Why a book about internationalization, and how is it worth your investment of time?

- iOS internationalization information and resources are scattered among fantastic blog posts, tutorials, and book chapters. This book’s goal is to be a central repository for all things internationalized.
- You can generate goodwill with your customers. Customers appreciate the fit and finish effort to make it easy for them to understand your application by having it support their cultural norms.
- The numbers point to supporting more than a single market: 60% of iPhone users worldwide are not native English speaking, and 50% of the countries within the top 10 for downloads and revenue in the iOS App Store are non-English-speaking countries from Europe and East Asia. App revenue in Asia increased by over 150% year-over-year (2012 to 2013), while North America’s revenue grew by just over 45%. (Source: Distimo 2014)
- The extra resources needed to support internationalization are not out of budget. Most Objective-C classes include a specific “locale” property or method. By including that call, you’ve added incredible flexibility and support to your application with minimum developer effort. Localization of your application can be an initial resource hit, but after it is established, it becomes a straightforward process of supplying string tables for supported languages.

These are compelling arguments for budgeting time and resources to internationalizing your application!

Reader Expectations

This book does have a few expectations of you, with the biggest being an understanding of Objective-C. A basic list of what is expected follows:

- **Objective-C**—This is the programming language for iOS. This book assumes that you have a strong working knowledge of Object-C because it pulls from existing classes and references methods and properties from each. It is also assumed that you have experience working with view controllers, XIBs, and Storyboards.
- **Xcode 5.x and later**—Much of the coding throughout the book references Xcode, its layout, and its tools and options, including Interface Builder. It is also assumed that you have a working knowledge of and familiarity with the iOS Simulator.
- **iOS 7 and later**—You will not need an iOS device, be it an iPhone or an iPad, but you do need to have experience using the device. Many of the instructions throughout the book are detailed, and a working knowledge of the System would be to your benefit.
- **Apple Developer account**—This is not a requirement but it’s great to have access to the Apple Developer resources and WWDC materials and videos. This also makes it easier to follow Chapter 7, “Submitting Your App,” and its discussion on localizing your app summary in the App Store.

How This Book Is Organized

Here’s a summary for each of the book’s chapters:

- **Chapter 1, “International Settings”**—This chapter covers the supported languages, regions, and calendars available in iOS 7. We’ll talk about what differentiates a “language” setting from a “region” or “locale” setting. This chapter goes step-by-step in how to change these settings. It gives details on how the settings are presented—in the native language for the most part—as well as background information on each of the settings. The chapter also includes detailed lists of the supported formats—date, time, currency, quotation marks, separators—for each language and region.
- **Chapter 2, “Characters and Encoding”**—This chapter covers character sets, understanding and working with them. We’ll talk about their storage and display and how encoding can potentially affect both of those aspects. Unicode is covered, as well as how it simplifies working with character sets. We’ll look at locale-specific character sets and their interaction with characters and ligatures. We’ll go into character details on retrieving their Unicode code points and the tools that make that possible. A discussion on fonts wraps up the chapter.

- **Chapter 3, “Coding for Locale”**—This chapter includes lots of code. There are many samples to cover specific locale classes, `NSLocale`, and the locale-specific arguments we can harness for our good from the `NSNumberFormatter`, `Address Book Framework`, `NSDateFormatter`, `NSTimeZone`, `NSDateComponents`, and `NSString` classes. We’ll demonstrate how they all respect the current locale settings for the System and verify that their return values are correct for the given locale.
- **Chapter 4, “Prepping Your App for Localization”**—We’ll take our lessons learned in Chapter 3 and apply them to a sample internationalization app, the `11&nExerciser`. This app will carry us through to Chapter 6. We’ll build an app that will display locale-specific information including date styles, character sets, measurement system, number formatting, sorting, current calendar, contact names, and many more items. No localization happens with this chapter. This is the prep, setting up our app to automatically handle localization.
- **Chapter 5, “Localizing Your App”**—We’ll take our sample app and walk through the localization process. We’ll cover the power of base localization, generate “dot-strings” files, and work with the key-value pairs in the “dot-strings” files. We’ll also cover localizing images and localizing the app name, and we’ll discuss working with translation services.
- **Chapter 6, “Adjusting the UI”**—The majority of this chapter covers Auto Layout and constraints. When the localized strings are applied, they can be longer than our default development language. We’ll cover how to prevent clipped strings with appropriate constraint settings. This chapter also covers how to support right-to-left languages via the proper constraint settings. We’ll also discuss implications of the height of the keyboard for different languages. We’ll cover available tools such as pseudo localization, double strings, and launch switches. We’ll wrap up with UI localization, images, colors, and string length.
- **Chapter 7, “Submitting Your App”**—This chapter hits on the App Store and iTunes Connect. It includes details on supported territories and pricing tiers. The chapter completes “language mapping” between the number of supported iOS 7 languages, to the languages spoken in iTunes Connect territories, to the languages listed in iTunes Connect on the app summary page. The chapter covers the how-tos for changing territories, as well as what fields to localize on the app summary page: name, description, keywords, URLs, screenshots, and EULAs. The chapter closes by covering Apple’s promotional materials and discussing how to see your app in regional App Stores.
- **Appendix A, “Case Study: *Boom Beach*”**—This appendix focuses on the numbers. Stats for a specific app’s download and sales increase after a Japanese localized version is provided.
- **Appendix B, “Web Resources”**—This is a listing of web resources related to internationalization and localization.

Getting the Sample Code

Chapters 4, 5, and 6 are written so that you can complete the sample project from scratch. Because they are set up with minimum instruction, I made the completed projects available on my GitHub account: <https://github.com/ShawnLa-18n>. Each chapter has its own associated project so that you can follow the project progress, see the mistakes, and see how they're fixed.

Contacting the Author

Feel free to contact me via e-mail at shawnlai18n@gmail.com if you have any comments or questions about this book, or contact me at the GitHub repository.

Characters and Encoding

Reading asks that you bring your whole life experience and your ability to decode the written word and your creative imagination to the page and be a co-author with the writer, because the story is just squiggles on the page unless you have a reader.

Katherine Paterson

Characters. Letters. Symbols. Items used on a printed page and on a multitude of displays we use today. When used in an organized sequence, they are interpreted to give meaning or, in other words, create words. Languages use different characters with accent marks and pronunciation marks to accentuate or provide meaning. We'll talk about what's involved in creating characters, things like *diacritics* and *surrogate pairs* and *ligatures*, and storing those characters (*encoding* and *code points*).

Chapter topics include the following:

- What's behind the scenes with characters
- How characters are stored and accessed by the OS
- How the OS determines what character to use based on its language setting
- How glyphs allow us to have different renditions or renderings of the same character
- What causes "garbage" characters or empty box characters to display

We'll hit the essentials about characters, strings, encoding, Unicode, and glyphs, and wrap up with fonts.

Characters

What is a character, what constitutes a character, and how is that character represented as far as the computer is concerned? A character is the smallest component of a written language that has semantic value. Focusing on the English (U.S.) alphabet, it's composed of 26 characters, and depending on the order and combination of those 26 characters, words can be formed, returning even more meaning. This section discusses characters in generic terms to get you into the mind-set of thinking of individual characters. The other goal I have is to make you aware of the different “characteristics” of characters. How characters are handled at the operating system level is covered in the “Code Pages and Encoding” section.

Types of Characters

You will be working with more than the characters from the English (U.S.) alphabet, so let's talk about characters that exist in other languages.

Accented Characters

Often, accents on characters such as the *acute* (´) accent and the *grave* (`) accent are referred to as *diacritical marks*. Other accents from European languages include the circumflex (^), umlaut (¨), and cedilla (,).

The main use of accents is to change the accented character's sound value. English examples include *naïve* and *Noël*. The accented characters (*diereses* in this case) show that these characters (vowels) are pronounced separately from the preceding vowel.

Acute and grave accents can indicate that a final vowel is to be pronounced, such as the French words *résumé* or *été*.

Accents can perform other functionality with different alphabetic systems. The Arabic harakat and the Hebrew niqqud systems are used for indicated vowel and tone sounds that are not conveyed through the basic alphabet. The Arabic sukūn and the Indic virama both designate the absence of a vowel. Special characters exist to mark for abbreviations or acronyms—as in the Cyrillic titlo and the Hebrew gershayim. The Greek language includes accents to indicate that letters of the alphabet are being used as numerals. In the Chinese Hanyu Pinyin system, accents are used to mark syllable tones in which the marked vowels occur.

Heads Up!

Different sounds can provide different meanings. You need to know you're using the correct character to give the correct intended meaning to your customer!

Chinese Characters

Chinese characters in themselves do not make up an alphabet. The writing system for Chinese is *logosyllabic*, meaning that a character generally represents one syllable of spoken Chinese and might be a word on its own or a part of a polysyllabic word. Chinese characters are all

derived from several hundred simple pictographs (representing physical objects) and ideographs (representing pronunciation or abstract notations).

Some Chinese characters have been adopted as part of the writing systems of other East Asian languages, such as Japanese and Korean. International software support for the Chinese, Japanese, and Korean languages is often shortcut as *CJK*. Table 2.1 shows examples of Asian characters.

Table 2.1 Sampling of Asian Characters

Language	Character	English Translation
Chinese	树	“tree”
Japanese	魚	“fish”
Korean	책	“book”

Characters for the Japanese language are usually a mixture of Chinese characters, or kanji, plus two syllabic scripts. At times the English (U.S.) alphabet is used as well. Having a working knowledge of 2,000 kanji characters is sufficient to read and comprehend most Japanese text.

Korean characters come mainly from an alphabetic script, Hangeul. Some hanja Chinese characters are used, but to a much lesser extent than with Japanese. When reading older Korean texts, an understanding of about 2,000 hanja characters is essential.

Table 2.2 contains different types of characters that are not necessarily found in any language’s alphabet but are interesting nonetheless. We’ll go over these types of characters and how to use and access them in the section, “Unicode and Encoding.” Punctuation characters have the capability to accentuate meaning, context, and understanding of text, but they do need to be associated with characters and words to accomplish that task. They cannot add meaning or understanding by themselves.

Table 2.2 Sampling of “Other” Characters

Category	Character	English Description
Punctuation	¶	Paragraph symbol, or pilcrow sign
Pictographs	☎	Snowman
Math Symbols	∩	Intersection
Letterlike Symbols	°C	Degree Celsius

Strings

Why do we use the term “strings,” and where did this term originate? Think of pearls, beads, or the like strung on a cord—something sequential connected in a line or arranged in a series or succession.

In general computer science terms, a string is traditionally a sequence of characters. This sequence of characters can be stored as a variable. Strings are generally treated as data types and are often implemented as an array that stores the characters using a manner of character encoding. The term *encoding* here refers to converting a character to its internal code point representation. Encoding is covered in detail in an upcoming section of this chapter.

In Objective-C, the string class is `NSString` and is the basic tool for representing text within your application. The `NSString` class provides powerful and flexible methods for manipulating its contents, as well as searching. And to add to its coolness, this class has native Unicode support.

Note

We will hit the `NSString` class and its other NS cousins in Chapter 3, “Coding for Locale.” I do want to call out this topic now because in the next section, “Code Pages and Encoding,” I make references to `NSString` objects, but again, the next chapter calls it out in greater detail.

Code Pages and Encoding

The most basic of definitions for character encoding is the assigning of a numeric code to a character. This particular number is called a *code point*. The OS represents these assigned code points by one or more bytes. This coding is a set of mappings between the bytes representing the numeric code used by the OS and the characters in the coded character set. This gives the OS a way to reference all available characters. If the encoding key is not available, potentially a different character is referenced, and the resulting data looks like garbage to the customer. To add to the complexity, there are many character sets and character encodings, giving us many ways to map among bytes, code points, and characters. Code samples in upcoming sections demonstrate this in action.

But where did this complexity originate? If we’re talking about characters—which are typically stored in one or two bytes—and numeric codes assigned to these characters, then why is there not a one-to-one correspondence? Let’s all sit back, relax, and enjoy a small history lesson on encoding.

ASCII Character Set

Back when the IBM-PC was first introduced—the Stone Age in computer time—due to localization being a lower priority, the characters having the highest importance were numbers, punctuation symbols, and unaccented English letters. All of them had a code associated with them, collectively called *ASCII* (American Standard Code for Information Interchange), which represented every character using a numeric value from 32 to 127. The capital letter “D” has a code point of 68 (decimal value), a lowercase “m” has a code of 77, and an exclamation point (“!”) has a code point of 33. All code points are conveniently stored in seven bits. Most systems at this time were using bytes of eight bits in length, so every possible ASCII character could be stored with a bit to spare. All code points below 32 were labeled unprintable and were used for control characters, such as 10, which is a “line feed,” and 13, which is a “carriage return.”

Extended Character Set

Noticing that bytes have room for a total of eight bits, people collectively got the idea, “Hey, codes 128 through 255 are available for our own aspirations.” One of the aspirations that came to be from this was the IBM-PC’s original equipment manufacturer (OEM) character set, which provided some support for European languages, specifically some accented characters, drawing characters including horizontal bars and vertical bars, and other characters.

After computers were purchased outside of the U.S., all manner of different OEM character sets appeared. All of these used the spare 128 characters for their own designs. Now what would happen in some circumstances was that a character that was encoded based on a different character set would appear as a completely different character on a computer using its own extended character set. For example, on some computers, the character code 130 would display as “é” but on computers sold in Israel the character would display as the Hebrew letter gimel (ג). When Americans would send their “résumés” to Israel, they would arrive as “רזסומזס.” In many cases, such as with Russian, there were many divergent ideas related to the upper 128 characters, which resulted in not being able to reliably interchange Russian documents.

ANSI Standard

Eventually, this OEM free-for-all got codified in the ANSI (American National Standards Institute) standard. With the ANSI standard, the consensus was to handle the characters below 128 the same as ASCII, and the handling of characters from 128 and up would depend on the locale. Code pages were established to handle these upper value characters.

The different ideas were codified into what are known as *code pages*. A code page is a table of values that describes a language’s encoding for a particular character set. Each of these code pages had a value associated with it. Greek speakers would use code page 737, Cyrillic speakers code page 855, and so on. All of these code pages were the same from codes 0 to 128, but different from codes 129 and up.

Asian Character Support: DBCS

This subject becomes even more complex when we're dealing with Asian character sets. Because the Chinese, Japanese, and Korean languages contain more than 256 characters, a different scheme needed to be developed, and it had to compete with the concept of code pages holding only 256 characters. The result of this was the double-byte character set (DBCS).

Each Asian character is represented by a pair of code points (hence the term double-byte), which allows for representing up to 65,536 characters. For programming awareness, a set of points are set aside to represent the first byte of the set and are not valued unless they are immediately followed by a defined second byte. DBCS meant that you had to write code that would treat these pairs of code points as one, and this still disallowed the combining of, say, Japanese and Chinese in the same data stream because depending on the code page, the same double-byte code points represent different characters for the different languages.

Heads Up!

Make every effort you can not to use the previously mentioned encodings. Beware, your app might need to deal with them when reading in a text file or accessing a Web site. Program defensively so that your code correctly handles this encoding. Coding examples in the upcoming sections demonstrate how to do this.

Unicode and Encoding

Hopefully, from what I presented in the preceding section, it is more than apparent just how nasty encoding can be, especially when you're dealing with code pages. Thankfully, the pain was felt far and wide, and the result was *Unicode*. Two of Unicode's mandates were to eliminate code page collisions and give each character its own individual code point value. The name "Unicode" comes from the desire to have a "universal" character set or, precisely, "universal code points."

Unicode Planes

Unicode is broken down into several planes, a *plane* being a continuous group of 65,536 (2^{16}) code points. Plane 0 is indicated as the *Basic Multilingual Plane (BMP)* and is where almost all of your day-to-day characters reside. The notable exception to this is the Emoji characters. Planes 1 through 16 are largely empty of characters and are termed *supplementary planes*. Table 2.3 lists the available Unicode Planes and the types of characters they hold.

Table 2.3 Unicode Planes

Plane	Character Types It Contains
Plane 0	BMP —Contains characters for most modern languages plus a large number of special characters. Code points in this plane are also used to encode CJK characters.

Plane 1	Supplementary Multilingual Plane (SMP) —Contains Emoji characters and other pictographs. Also holds historical scripts such as hieroglyphics.
Plane 2	Supplementary Ideographic Plane (SIP) —Contains CJK characters.
Planes 3–13	Unassigned —Temporarily named the Tertiary Ideographic Plane.
Plane 14	Supplementary Special-purpose Plane (SSP) —Contains nongraphical characters, such as those used for XML language tag characters, as well as alternative glyphs.
Planes 15–16	Supplementary Private Use Area-A and Area-B , respectively—Contains characters that are used internally by fonts for auxiliary glyphs and ligatures.

In simplest terms, Unicode provides a code point for every character or symbol in nearly all the world’s writing systems. Unicode code points are written in the form “U+#####” in which “#####” is made up of four to six hexadecimal digits. To give three quick examples, the code point U+0062 (decimal 98) represents a lowercase “b”—the same character and same value in the Latin ASCII table. The Cyrillic capital letter “de” or “Д” has the Unicode code point of U+0414 (decimal 1044), and the fleur-de-lis symbol “♣” is U+269C (decimal 9884).

To provide room for 65,536 characters, Unicode was originally conceived as a 16-bit encoding. This provided enough space to encode all modern scripts around the world. Private Use areas were designated to hold rare or obsolete characters. Unicode has approximately 10% of its total available code points in use, leaving ample room to grow.

Combining Character Sequences

Certain characters can be represented either as a single code point or as a sequence of two or more code points. Take, for example, the “i” character. This can be represented either as a single-character “i” (“Latin Small Letter I with Grave,” U+00EC), or as a combination of two characters, “i” (“Latin Small Letter I,” U+0069) and “˘” (“Combining Grave Accent,” U+0300). Both of these forms are variants of a composite or combining character sequence. This combination of characters is not restricted to Latin scripts but includes CJK character sets as well. With Hangul, the syllable ㄱㅏ can be represented as a single code point, U+AC00, or as the sequence ㄱ + ㅏ, U+1100 and U+1161.

As far as Unicode is concerned, the two characters are not equivalent—because they contain different code points—but they do have *canonical equivalency*. In other words, they have the same appearance and meaning. See the “Diacritics” section later in this chapter for examples of this combination in action.

Duplicate Characters

As you look through existing Unicode tables, do you see double? The characters might look the same, but that’s the only similarity they have. The display might be identical, but some characters are encoded at different code points to retain the character’s meaning. Take, for

example, the Latin character “A” (U+0041). Its shape is identical to the Cyrillic “A” (U+0410), but they are two very separate characters. Having the separate code points also simplifies the conversion from legacy encodings.

Of course, there is the contrary scenario in which there truly are duplicate characters, both in display and in the character’s meaning. An example here would be the Angstrom sign, “Å.” This character owns two listings; the first has the character info “Latin Capital Letter A with Ring Above” (U+00C5), and the second has “Angstrom Sign” (U+212B).

Other characters that fall into this category have a property known as *compatibility equivalence*. Compatibility represents essentially the character but a slightly different visual appearance and a different behavior for this character. Concrete examples include Greek letters, which can be mathematical and technical symbols, Roman numerals, or actually Greek text.

Other examples of compatibility equivalence include ligatures. The single character “ff”—having character info of “Latin Small Ligature FF” (U+FB00) is compatible with the sequence of two characters “f” having character info of “Latin Small Letter F” (U+0066) + “Latin Small Letter F” (U+0066). They might render and display similarly, but that similarity is dependent on context, typeface, and the text renderer.

Heads Up!

If your sorting is not giving the expected results, check your characters and make no assumptions. You could be working with a character that is identical to another character encoded with a different code point.

Encoding in Action

Wow. I’m thankful we have Unicode, and although it’s imperfect, it is a far, far more direct approach to working with characters and handling encoding. Let’s move on to seeing encoding in action with some code samples and then wrap up this section with some encoding bloopers. We’ve already talked about ASCII encoding; now let’s look at encoding definitions for two of the most common encoding formats, UTF-8 and UTF-16.

UTF-8

UTF-8 stands for Universal Character Set (UCS) Transformation Format—8-bit.

Universal Character Set and Unicode

UCS and Unicode have a relationship and distinctions as well. UCS and Unicode are related by the fact that Unicode is regarded as the 16-bit coding of the Basic Multilingual Plane (BMP) of the UCS.

The UTF-8 format uses variable-width encoding and is capable of storing and representing every character in the Unicode character set. Its design was based on avoiding endianness

complications and byte order marks found in the UTF-16 and UTF-32 encoding formats and, even more important, backward compatibility with the ASCII format (see the “Endianness” note later in the chapter for more detail about that and byte order). This encoding format accounts for more than half of all web pages, and the Internet Mail Consortium recommends that all email programs create and display messages using UTF-8. It’s increasingly becoming the default character encoding in software applications, operating systems, and programming languages. Xcode is a prime example of this, as shown in Figure 2.1.

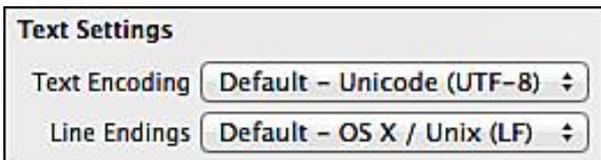


Figure 2.1 The default encoding for source files in Xcode.

UTF-16

UTF-16 is the character encoding capable of encoding well over one million code points in the Unicode code space. UTF-16 is short for 16-bit Unicode Transformation Format. The Unicode code space encompasses from code point 0 to 0x10FFFF. Its encoding is variable-length; code points are encoded with one or two 16-bit code units. UTF-16 encoding provides excellent support for Asian languages. It is not, however, ASCII compatible.

Coding Encoding

Let’s look at some code samples.

Text comes down to the wire as an `NSData` instance in which the “wire” could be a network condition or a file I/O action. To encode the text you are accessing, you can allocate an `NSString` and initialize it via the `initWithData:encoding` method. Notice the second parameter: `encoding`! So you need to know how the text was encoded.

Example 1: Encoding to ASCII

Listing 2.1 takes a string that contains accented characters, in this case the German word “Fußgängerübergänge.” We’ll examine what characters get “lost” from this encoding.

Listing 2.1 Encoding Text to ASCII

```
// Fußgängerübergänge - "sea voyage"
NSString *uberUmlats = @"Fußgängerübergänge";
NSData *ASCIIData =
```

```
↳[uberUmlats dataUsingEncoding:NSUTF8StringEncoding allowLossyConversion:YES];
NSString *encodeToASCII =
↳[[NSString alloc] initWithData:ASCIIData encoding:NSUTF8StringEncoding];
NSLog(@"Encoded to ASCII - %@", encodeToASCII);
```

The encoding that is in place is ASCII via the `NSASCIIStringEncoding` specifier. The reason we get `Fusgangerubergange` as our return value is two-fold. First, by specifying ASCII we are limited to characters with code point values under 128, so essentially no accented characters, of which our string has three, “ß,” “ä,” and “ü.” Second, by using `NSString`’s `dataUsingEncoding:allowLossyConversion:` instance method, we can specify, in a sense, “Handle all the characters I give you, and if it’s an accented character, I’m okay with your losing that accent.” Although the result is not a correct German word, its display is very close, and its meaning can reasonably be interpreted. If we change the encoding type to `NSMacOSRomanStringEncoding`, there’s no guarantee how the characters will be converted and encoded. In fact, with this encoding, the result is an unintelligible `Fu$g?¿nger¿berg¿nge`.

Example 2: Returning the ASCII Value from a Character

Listing 2.2 takes a single character as its argument and returns the ASCII value associated with it.

Listing 2.2 Returning the ASCII Value for a Character

```
NSString *encodingFun = @"a";
if ([encodingFun length] > 0) {
    unichar ASCIIValue = [encodingFun characterAtIndex:0];
NSLog(@"ASCII value is %d", ASCIIValue);
}
```

The returned ASCII value for the character `a` is 97. Note that the type `unichar` is used because it is a typedef for an unsigned short. The value returned by the `characterAtIndex` method is the Unicode decimal representation for the code point. An `NSString` object is usually represented by an array of `unichars` internally, hence the reason we are using `unichar` as a return type.

Example 3: Encoding an ASCII String to UTF-8

Listing 2.3 shows the potential of repairing some “damage.” Typically, if you are working with text that has accented characters, those characters are misinterpreted when encoded to ASCII. This example starts with a misinterpreted string and correctly encodes it to UTF-8. Note that a little magic incantation is involved with this snippet because we need to take an `NSString` object and covert it to a plain C string.

Listing 2.3 Encoding an ASCII String to UTF-8

```
NSString *notUTF = @"N\u00fcrnberg";
NSString *nowUTF =
↳ [NSString stringWithUTF8String:[notUTF cStringUsingEncoding:
NSMacOSRomanStringEncoding]];
NSLog(@"Now a UTF8 string: %@", nowUTF);
```

The returned value is Nürnberg.

Example 4: Returning a String from an Encoding URL

Listing 2.4 takes an encoded string, in this case encoded from a valid URL, and returns text. With an encoded URL, many of the punctuation symbols and nonprinting characters are encoded, such as a space character encoded as %20. In the following argument, the chevrons < and > are encoded as %3C and %3E, respectively. The ampersand & is encoded as %26.

Listing 2.4 Returning a String from an Encoding URL

```
NSString *currentEncodedString = @"%3CTom%26Jerry%3E";
NSString *currentDecodedString =
↳ [currentEncodedString
stringByReplacingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
NSLog(@"Decoded string: %@", currentDecodedString);
```

The code returns <Tom&Jerry>.

Objective-C Encoding Enumerations

The constants listed in Table 2.4 are the string encodings provided by the `NSString` class. You'll use these encodings for the specified operating system so that the characters being encoded have the proper, understandable codes so that they are appropriately displayed. For example, if you are exporting text from your app that you know is going to a platform that does not support Unicode, you should encode it to the ASCII format using the `NSASCIIStringEncoding` constant.

Table 2.4 Encoding Formats

Enum Name	Value	Description
<code>NSASCIIStringEncoding</code>	1	Supports only lower-value, 0–127, ASCII characters. No support for higher-bit characters. You'll want to use this only if characters need to be in 7-bit ASCII.
<code>NSNEXTSTEPStringEncoding</code>	2	Supports encoding used by NeXT.
<code>NSJapaneseEUCStringEncoding</code>	3	Supports a variable-width encoding used to represent the elements of three Japanese character set standards (JIS X 0208, JIS X 0212, JIS X 0201).
<code>NSUTF8StringEncoding</code>	4	Allows for full Unicode support. Characters display as ASCII when the text is plain English.
<code>NSISOLatin1StringEncoding</code>	5	Encoding for European characters as high-bit ASCII (values 128–255). Also cross-platform international standard. Very common for Web sites to deliver their text in this format.
<code>NSSymbolStringEncoding</code>	6	Way for symbol characters to be encoded.
<code>NSNonLossyASCIIStringEncoding</code>	7	Similar to <code>NSASCIIStringEncoding</code> . Reinforces no high-bit characters.
<code>NSShiftJISStringEncoding</code>	8	Supports Shift Japanese Industrial Standards encoding.
<code>NSISOLatin2StringEncoding</code>	9	Supports ISO/IEC 8859 encoding of European languages.
<code>NSUnicodeStringEncoding</code>	10	Encodes all characters as two bytes. Marker bytes exist at the beginning of the file stream to signify whether the byte order is little or big endian.
<code>NSWindowsCP1251StringEncoding</code>	11	Supports Cyrillic text. This is the same as <code>AdobeStandardCyrillic</code> .
<code>NSWindowsCP1252StringEncoding</code>	12	Supports “WinLatin1” encoding. This is the most common for European/English text. Windows boxes that host Web sites deliver their context in this format.
<code>NSWindowsCP1253StringEncoding</code>	13	Supports “Greek” encoding.
<code>NSWindowsCP1254StringEncoding</code>	14	Supports “Turkish” encoding.

<code>NSWindowsCP1250StringEncoding</code>	15	As with Latin1, supports “WinLatin2” encoding.
<code>NSISO2022JPStringEncoding</code>	21	Japanese encoding for e-mail.
<code>NSMacOSRomanStringEncoding</code>	30	Default encoding for the Mac.
<code>NSUTF16StringEncoding = NSUnicodeStringEncoding</code>		Supports UTF16. <code>NSUnicodeStringEncoding</code> is an alias for <code>NSUTF16qStringEncoding</code> .
<code>NSUTF16BigEndianStringEncoding</code>	0x90000100	<code>NSUTF16StringEncoding</code> encoding with explicit endianness specified.
<code>NSUTF16LittleEndianStringEncoding</code>	0x94000100	<code>NSUTF16StringEncoding</code> encoding with explicit endianness specified.
<code>NSUTF32StringEncoding</code>	0x8c000100	Convert <code>NSString</code> to <code>NSUTF32StringEncoding</code> .
<code>NSUTF32BigEndianStringEncoding</code>	0x98000100	<code>NSUTF32StringEncoding</code> encoding with explicit endianness specified.
<code>NSUTF32LittleEndianStringEncoding</code>	0x9c000100	<code>NSUTF32StringEncoding</code> encoding with explicit endianness specified.
<code>NSProprietaryStringEncoding</code>	65536	As the name of the enum infers, used for custom, proprietary encodings.

Endianness

We won’t go into detail in this book about string support, but I do want to call out endian. As you can be confident, this is neither a reference to a citizen of the country of India nor a reference to a Native American person, but rather a reference to the order in which bytes are stored in memory. Little endian stores data with the least significant byte in the smallest address. Big endian stores the most significant byte of a data word in the smallest memory address and the least significant byte in the largest address. The iOS operating system stores data in the little endian format.

Encoding Gone Bad

Table 2.5 shows an example of some “bad” encoding. (Is it true that there’s no bad encoding, just bad programmers?)

Table 2.5 Examples of Encoding Gone Bad

Original Text	After Encoding	Notes
der Mülleimer	der MÃ¼lleimer	Stored UTF-8 being interpreted as Latin-1.
Nürnberg	N√rnberg	UTF-8 to Mac Roman.
Understanding how to “get” certain characters, certain glyphs and explain how they’re represented	Understanding how to ç\$B!FçBgetç\$B!Gç(B certain characters, certain glyphs and explain how they_ç\$B!GçBre represented	Source file was Japanese (ISO 2022-JP) and was read in using the UTF-8 encoding.

Heads Up!

If your app needs to include a text file for an end-user agreement or licensing documentation, verify that it is properly UTF encoded. Ensure that any “read me” file that you will be internationalizing supports all the characters of the region it is shipped to. It is far too easy to look foolish with missed encoded characters in potentially the *first* file a customer reads!

Diacritics

A diacritic, or *diacritical mark* is a mark, point, or sign attached to a character to distinguish it from another of similar form. This mark can also give that character a particular phonetic value to indicate stress. A *cedilla* (hook or tail “ç”) accomplishes this when it is added under certain letters to modify their pronunciation: “ç ð à È Û.” Other diacritics that affect a character’s pronunciation include the tilde “~” and the circumflex (chevron-shaped “^” in the Latin script), or the macron when it is placed above a vowel, as in “ē Ä.” You will hear diacritics referred to as *combining characters*.

Diacritics can be treated in several ways:

- A diacritic can be a Roman base character plus a diacritic character. A combination such as “á” might be encoded either as a single character with character info of “Small A With Acute,” or as a sequence of characters, “Small A” + “Combining Acute.” Another example is the character “Ä” represented as the Unicode code point U+00C4, or as a pair of code points, U+0041 and U+0308.
- The “Small E with Grave,” or “è” character, might be rendered via glyphs—either a single composite glyph, or using two separate glyphs, one for “e” and another for an overstriking grave accent.
- In some *orthographies* (the relationship between sounds and letters), the combination “è” would be considered a *grapheme* (a letter or a number of letters that represent a *phoneme*, or speech sound that distinguishes one word from another), whereas in other orthographies “e” and the grave accent would each be considered a grapheme in a word.

In other words, if an orthography has an “è” as a grapheme, then it should be encoded as a single character and an associated single glyph. On the flip side, if an orthography has separate graphemes for the “e” and the “˘” (the grave accent), they should be encoded as separate characters and rendered as separate glyphs.

Precomposed Diacritics

We’ve been talking about diacritics and how they can be added to or combined with other characters. Some of the terms given to this combination are *composite character*, *decomposable character*, and *precomposed character*. Let’s look at an example to see why this distinction is important. The character “ñ” is a precomposed character because it is treated as an individual Unicode character and has a Unicode code point of U+00F1. Technically, this character can be decomposed into an equivalent string of a base character “n” (U+006E) and a combining tilde “~” (U+0303). Precomposed characters are a solution for handling legacy support of special characters in character sets. They are included for the primary reason of aiding systems with incomplete Unicode support in which the individual decomposed characters can be rendered successfully.

In looking at our “Small Letter N with Tilde” example, we could potentially be dealing with one single character or two individual, separate characters. If our code is doing any kind of character or string comparison, it is possible to have a test fail. To ensure that the expected single characters are used in the comparison, Unicode normalization is required. This can be accomplished via the `precomposedStringWithCanonicalMapping` method. Let’s look at some code. In Listing 2.5, we’ll work with our “Latin Small Letter N with Tilde” as a combined character and compare it to the precomposed character.

Listing 2.5 Displaying Combined and Precomposed Characters

```
NSString *combinedCharacter = @"n\u0303";
NSString *precomposedCharacter = @"ñ";
BOOL isEqual = [combinedCharacter isEqualToString:precomposedCharacter];
NSLog(@"The 'combined' character, '%@', is %@ to 'precomposed' character, '%@'",
combinedCharacter, isEqual ? @"equal" : @"not equal", precomposedCharacter);
```

This returns “ñ is not equal to ñ.”

Now applying the same test, but first normalizing the characters, we use this:

```
NSString *combinedNormalized =
↳[combinedCharacter precomposedStringWithCanonicalMapping];
NSString *precomposedNormalized =
↳[precomposedCharacter precomposedStringWithCanonicalMapping];
BOOL isEqualNorm = [combinedNormalized isEqualToString:precomposedNormalized];
NSLog(@"The 'combined-normalized' character, '%@', is %@ to 'precomposed-
normalized' character, '%@'", combinedCharacter, isEqualNorm ? @"equal" : @"not
equal", precomposedCharacter);
```

This returns “ñ is equal to ñ.”

Heads Up!

Be aware of the gamut of diacritics that exist, their effect on line spacing and line height, and if they are not precomposed, their effect on the total number of characters in a line of text. If you are working with custom fonts in your app, be certain that the font contains the glyphs of the diacritics you need.

Surrogate Characters

Surrogate characters are typically referred to as *surrogate pairs*. They are the combination of two characters, containing a single code point. To make the detection of surrogate pairs easy, the Unicode standard has reserved the range from U+D800 to U+DFFF for the use of UTF-16. No characters are assigned to code point values in this range. When programs see a bit sequence that falls in this range, they immediately—zip! zip!—know that they have encountered a surrogate pair.

This reserved range is composed of two parts:

- **High surrogates**—U+D800 to U+DBFF (total of 1,024 code points)
- **Low surrogates**—U+DC00 to U+DFFF (total of 1,024 code points)

A lone surrogate is invalid in UTF-16; surrogates are always written in pairs, with the high surrogate followed by the low.

With UTF-16 encoding, characters with code points in ranges U+0000 through U+D7FF and U+E000 through U+FFFD are stored as single 16-bit units.

Table 2.6 contains examples of surrogate pairs.

Table 2.6 Examples of Surrogate Pairs

Character	Code Point	Surrogate Pair
	U+10000	{U+D800, U+DC00}
	U+10E6D	{U+D803, U+DE6D}
	U+1D11E	{U+D834, U+DD1E}
	U+10FFFF	{U+DBFF, U+DFFF}

The following code snippet shows you how to get a printout of a surrogate pair when you are given its code point:

```
uniChar characterArray[2];
CFStringGetSurrogatePairForLongCharacter(0x10FFFF, characterArray);
NSString *surrogate = [[NSString alloc] initWithCharacters:characterArray length:2];
NSLog(@"Surrogate: %@", surrogate);
```

Note that this is taking advantage of the `CFStringGetSurrogatePairForLongCharacter` function, which maps a UTF-32 character to a pair of UTF-16 surrogate characters. We need an array to plug the resulting UTF-16 pair into—that’s what the `characterArray` is for—and then the `initWithCharacters:length:` method of `NSString` does the rest.

Heads Up!

In speaking to fellow developers, I’ve found that one of the “gotchas” they’ve had to debug and fix was from the result of copying and pasting surrogate pairs. They’ve also had issues when working with Greek/mathematical symbols. The result of the paste action was munged and missing characters. Do thorough testing with a range of characters, those with high Unicode code point values, surrogate pairs, and even Emoji characters, especially if your app supports text entry.

Emoji

I’m making a special callout on the Emoji characters because they are extremely popular, and Apple both uses a special font to represent them and provides a keyboard to input just Emoji characters.

Introduced in the late 1990s from a Japanese mobile phone provider, Emoji is the Japanese term for *picture characters*. Created by Shigetaka Kurita as an effort to retain his company's customer base, the smiley-faced icons gave their text messages more cuteness. The other supporting factor of the Emoji characters was the ability to give contextual information with a single character. What's the weather going to be like today? That's easily presented with a sun or umbrella or cloud Emoji character.

Figure 2.2 shows the first page of the Emoji keyboard.



Figure 2.2 The Emoji keyboard.

Apple Color Emoji is a font available on both iOS and OS X to provide support for the Unicode Emoji characters. Instead of this font having glyphs with black and white outlines, it has full-color, higher-resolution images for each of the nearly 900 glyphs it supports.

Strong support of Emoji has been a hard target to hit because it has historically occupied a private use area of Unicode with a range of code points from U+1F604 to U+1F539.

Retrieving Characters from Unicode Code Points

A quick way to see what character is associated with a given code point is via the Mac OS Calculator app. After the app is launched, if you switch the view to programmer, you can enter

hex values and the app will return the available ASCII or Unicode character. If no character is available, nothing shows on its “screen.” See Figure 2.3 for a screenshot of the Calculator app.

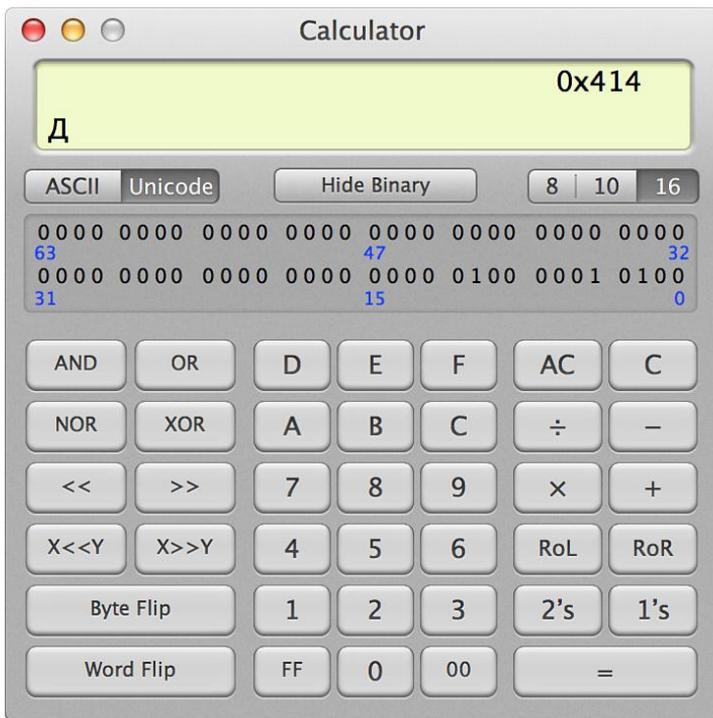


Figure 2.3 The Mac OS Calculator app in programmer view, displaying the Cyrillic “de” character.

Here, I simply entered “414” for the Cyrillic capital character “de” (“Д”), mentioned in an early section. The binary representation of the code point value is displayed as well. This is extremely useful if you have the code point number, but what if you do not?

Obtaining Unicode Code Points

I have created an iPhone project available on my GitHub page that will return the Unicode code point value for a given character. The project also returns the ASCII value, when available, for the provided character. By having the code point for a character, you have more reference information. You’re more easily able to search for the character and reproduce it, and then test it against a required font to ensure that a glyph for that character exists in the font set. You will also know what the impact on encoding this character will be. By having the code point value, you don’t need to keep the actual character saved in a file and then load the file and copy and paste the character.

The workflow of my project is this:

- You see a character from a text or Web site or even a file you've opened on your device.
- You select the character and then copy it.
- You switch to my project and paste it into the “String” field.

The GitHub project link is <https://github.com/ShawnLaAppleDev/UnicodeValueGrabber>.

The code takes advantage of both the `NSString` class method `initWithString` and the format specifier `%04x` to return the Hexadecimal value of supplied character. Note that the format specifier is set up to have four placeholders that include leading zeros (see Listing 2.6).

Listing 2.6 Returning the Unicode Code Points

```
unichar ch = [unicodeString characterAtIndex:0];
NSString *unicodeHex =
↳ [NSString stringWithFormat:@"%U+%04x", ch] uppercaseString];
return unicodeHex;
```

Note

You can dig even more deeply into Unicode at www.unicode.org.

Glyphs

It is easy to confuse “glyphs” with “characters” because it is the glyph of the character that is drawn onscreen and hence what we are looking at. A glyph is a pattern, a shape, or an outline of the character’s image. Characters are what you type; glyphs are what you see.

Two points need to be called out:

- A character conveys differences in meaning or sound. No appearance property is associated with it.
- A glyph conveys differences in appearance. The key thing is appearance. A glyph has no intrinsic meaning.

Figure 2.4 shows a sampling of some glyphs for the Latin character “c.” Note that they are all of the same character, lowercase “c,” and therefore have the same meaning, but they are displayed, shaped, and outlined differently.

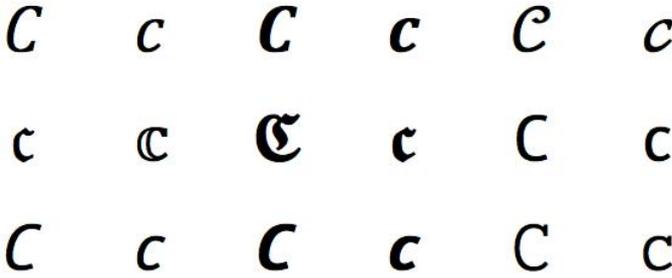


Figure 2.4 A sampling of glyphs for the lowercase character “c.”

There are also cases in which a character will have a glyph assigned to it based on the font set so that the shape it displays is nothing like the traditional shape of the character. “Wingdings” and assorted symbol fonts like that are prime examples in which the lowercase “c” character actually displays as ™.

Also, variations in a character’s glyph can be associated with things like cursive connectors. In this scenario, the character is displayed traditionally, but it has slight changes depending on what character precedes it and what character follows it. Again, no meaning is lost; we have just gained flourishes.

Contextual Glyphs

For the Arabic and Indic family of languages, a character’s glyph can change greatly depending on the glyph’s position within the word and can change depending on characters that follow and precede it. Let’s focus on the Arabic character “Ain,” “ع” (Unicode code point U+0639). Figure 2.5 displays the different glyphs used for “Ain” depending on its context and position.

Position in Word	Isolated	End of a Word (Final)	Middle Form (Medial)	Beginning of Word (Initial)
Glyph Form	ع	ع	ع	ع
Sample Text		إصبع	سعر	عدن
English Translation		Finger	Price	Aden

Figure 2.5 Examples of contextual glyphs.

Arabic is a right-to-left language, so the first/initial character is the rightmost, the second is to the left of that, the third is to the left of the second, and so on.

Another example is the Greek character sigma (σ). When it is used at the end of a word and the characters of that word are not all uppercase, the final form of the character “ς” is used, for example, “Ὀδυσσεύς” (Odysseus). Note the two sigmas in the center of the name that remain the same and the word-final sigma at the end. Same character, different glyphs. This example also demonstrates that uppercase and lowercase are handled as separate characters and not as the same character—same character value, only *displayed* differently.

Chapter Quote

The quote at the beginning of this chapter comes from children’s author Katherine Paterson. I included it because she mentions “reading” and “decoding,” which are both file I/O functions we are covering. The mention of “squiggles” reminds me of glyphs.

Now that we’ve covered glyphs, let’s move on to fonts.

Fonts

The term *font* is a common, everyday household term. We generally think of it as the shape and display of the characters we are working with, as well as the size and spacing. It is a combination of these properties, as well as the typeface associated with the font.

Font files are your storage depot for the glyphs that are associated with the characters. Well-crafted fonts won’t fake bold, italic, and bold-italic variations but have built-in, designed glyphs for these variations. After your application has worked out what characters it is dealing with, it will look in the font for glyphs in order to display or print those characters. Of course, if the encoding information was wrong, it will be looking up glyphs for the wrong characters.

A given font will usually cover a single character set. In the case of a large character set, like Unicode, just a subset of all the available characters will be available. This is one of the many reasons you will see specific fonts for CJK characters. It is more practical to have specific fonts hold specific character sets for both performance and file-size benefits.

If your font doesn’t have a glyph for a particular character, some applications as well as the OS will look for the missing glyph in other fonts on your system. Although this eliminates a missing glyph from displaying as an empty box or a box containing a question mark, it does have the potential of having the glyph look different from the surrounding text, like a ransom note.

Heads Up!

Including fonts will cause the bundle size to grow. With iOS, any custom fonts and font files your project needs must be included in the application bundle. Currently, there is not an option to install fonts to a Library/Fonts type folder on an iOS device. Be aware of how this will affect your bundle size and, by association, your download times.

Ligatures

The term *ligature*, which simply means “connection,” originates from the Latin *ligari*. The term itself doesn’t imply a certain purpose or use. Today, there are two possible ways to define a ligature, and both ways can appear in connection or individually. If we talk about the display of characters, a ligature is made from two or more letters, which appear connected. In handwriting such connections are created all the time, especially with cursive print.

Some ligatures are two separate characters displayed with a connected glyph, whereas the glyph is one character with its own code point.

Standard ligatures might include *fi*, *fl*, *ff*, *ffi*, *ffl*, and *ft*. The purpose of these ligatures is to make certain letter parts that tend to knock up against each other more attractive.

Here are some individual Unicode ligature characters:

- **æ**—CYRILLIC SMALL LIGATURE A IE; Unicode: U+04D5; UTF-8: D3 95
- **fl**—LATIN SMALL LIGATURE FL; Unicode: U+FB02; UTF-8: EF AC 82
- **fi**—LATIN SMALL LIGATURE FI; Unicode: U+FB01; UTF-8: EF AC 81

Code Snippet to Compare Ligatures

Listing 2.7 compares the single-character ligature “ff” to the two-character equivalent “ff.” The `localizedCompare` method returns an `NSComparisonResult` value, which could be an enum of `NSOrderedAscending`, `NSOrderedSame`, or `NSOrderedDescending`.

Listing 2.7 Using Localized Compare to Determine Whether Characters Are Equal

```
NSString *characters = @"ff"; // Two "f" characters
NSString *ligature = @"\uFB00"; // Single character - "ff" ligature
NSComparisonResult result = [characters localizedCompare: ligature];
if (result == NSOrderedSame){
    NSLog(@"%@ is equal to %@", characters, ligature);
} else{
    NSLog(@"Characters are not equal.");
};
```

The code returns “ff is equal to ff.”

Summary

This chapter talked about characters and what kinds of “characters” they can be. We covered the history of encoding, starting with ASCII’s use of code pages and how that was fraught with danger. From there we moved to encoding and working with code points for characters and what can happen to a character if the wrong encoding is used. Hopefully, that discussion “scared you straight” so that you’ll use Unicode code points and use them often. From there, we talked about different ways Unicode handles characters, including combining, precomposed, surrogate, and duplicate. Glyphs, ligatures, and fonts wrapped up the chapter when we covered how they affect the display of the character.