Internationalization Programming Topics

Internationalization > Localization

2009-01-06

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to Internationalization Programming Topics

Today's applications are marketed to a global audience. Selling your applications to that audience requires the customization of your software for each target market. Users in a foreign country are not going to want a user interface in a language they do not understand. Similarly, there may be images that you find acceptable but which are considered quite rude in other cultures. The problem is how do you create software in a language that you do not understand? The answer is through the internationalization technologies found in Mac OS X.

Rather than rewrite your software for each language you want to support, you can internationalize it to support any language. This process involves separating any user-visible text and images from your executable code. Once you have this data isolated into separate resource files, you can translate it into the desired languages and integrate the localized resource files back into your application bundle. This document helps you understand the steps needed to prepare your application for these processes.

Organization of This Document

This document includes the following articles:

- "Internationalization and Localization" (page 11) introduces the process and terminology associated with internationalization and localization.
- "Support for Internationalization" (page 13) describes the support for internationalization provided by Mac OS X.
- "Language and Locale Designations" (page 19) describes the conventions for identifying languages and locales in your application.
- "Guidelines for Internationalization" (page 25) provides tips to help you internationalize your software.
- "Getting the Current Language and Locale" (page 29) shows you how to find out which localization is currently in effect.
- "Preparing Your Nib Files for Localization" (page 33) provides tips for localizing your nib files including how to extract strings using `ibtool`.
- "Strings Files" (page 37) shows you how to create string resource files and get the resulting strings in your code.
- "Internationalizing Other Resources" (page 47) provides tips on how to localize programmatically-generated content using the current locale information.
- "Localizing Pathnames" (page 49) describes the Mac OS X support for localized bundle and directory names and shows you how to support this feature in your application.
- "Notes For Localizers" (page 53) provides tips for people who localize content in Mac OS X.
- "File Encodings and Fonts" (page 55) provides legacy information related to file encodings in previous versions of Mac OS, along with information about how to support those encodings in Mac OS X. It also describes some issues surrounding the use of fonts with different file encodings.

See Also

The bundle mechanism plays a prominent role in supporting localized versions of an application. In addition, part of the internationalization process involves using resource files instead of hard coding strings and other localizable content into your executable. You should therefore read the following books for related information about the internationalization process:

- *Bundle Programming Guide* provides information about the bundle structure of applications and how it supports localized content.
- *Resource Programming Guide* provides information about resource files and how you load them into your application.

Internationalization and Localization

Internationalization is the process of designing and building an application to facilitate localization. **Localization**, in turn, is the cultural and linguistic adaptation of an internationalized application to two or more culturally-distinct markets. When users launch a well-localized application, they should feel fully at home with it and not feel like it originated from some other country.

Internationalization and localization are complementary activities. By internationalizing an application, you create the programmatic infrastructure needed to support localized content. By localizing that application, you then add resources that are customized for people of a different culture. Localizing an application often involves translating the user-visible text of the application, but that is only part of the work that must be done. Other parts of your application must also be modified, including the following:

- Nib files (windows, views, menus) must be modified to accept translated text.
- Static text must be translated.
- Icons and graphics (especially those containing culture-specific images) must be changed to be more culturally appropriate.
- Sound files that contain spoken language must be rerecorded for each supported language.
- Online help must be translated.
- Dynamic text generated by your program (including dates, times, and numerical values) must be formatted using the current locale information.
- Text handling code must calculate word breaks using the current locale information.
- Tabular data must be sortable using the current locale information.

The process for internationalizing a Mac OS X application is easy and the same on both platforms. Inside your application's bundle directory, there is one file containing your application's executable code and potentially many separate files containing the localized resources. The resource files for a single localization are stored together in a language-specific project directory of the bundle. The bundle may contain any number of language-specific project directories, each one representing a distinct localization that the application supports. You can even store localizations for single-byte and double-byte languages together in the same bundle.

The first step in the development of an internationalized application is to identify all culture-specific information in your application. Scour your user interface for culture-specific text, images, and sounds and put them into resource files. As you do that, update the underlying code so that it loads the data it needs from those resource files. As you go through your code, you should also look for places where your user interface displays date, time and currency values and make sure you are formatting them using the current locale information.

Once your application's user interface is frozen (which may be before the code freeze), you can begin to localize it. For each localization, the localization team creates language-specific versions of the nib files, text, images, and sound files. If your application is properly internationalized, the process doesn't require modifications to your source code. You can therefore create the translations in-house or contract with an outside localization service.

Even if you do not plan to support multiple languages immediately, internationalizing your application up front is a good idea. There is also no penalty for building internationalization support into your application. Even if your application is monolingual, you still need to test all available code paths. If your code is properly internationalized for one language, it should require little additional testing to support other languages. In addition, storing resources outside of your executable file makes it easier to change the appearance of your application later without modifying the underlying code.

Support for Internationalization

Mac OS X supports internationalization and localized content in a variety of ways.

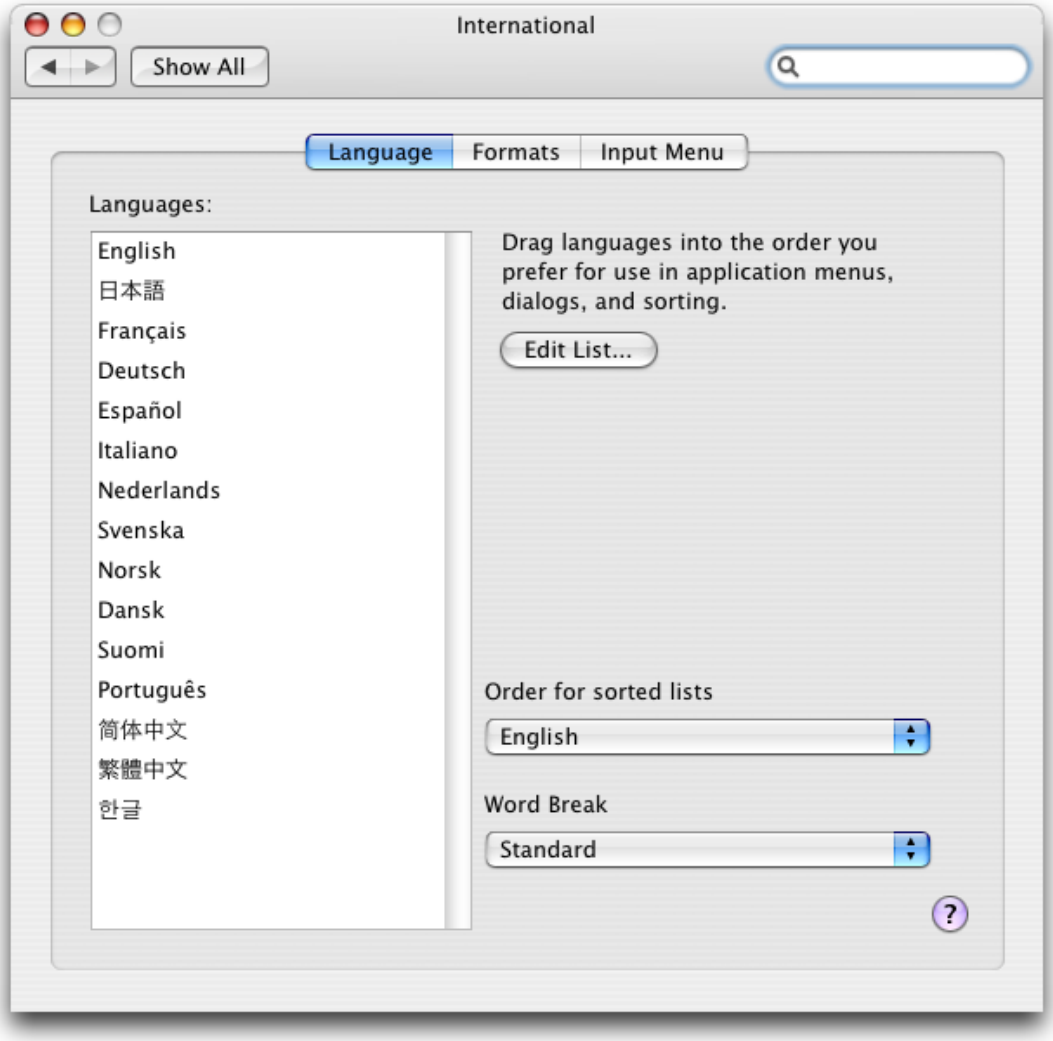
- It allows the user to specify a set of preferred languages.
- It provides a mechanism for storing multiple language-versions of resources within an application.
- It offers APIs and tools for generating string tables and for accessing localized strings.
- It allows additional languages to be easily added to an application, even at run time.
- Through Xcode, it provides project support for internationalization.

The notion of a preferred language set and the mechanisms of bundles, localized resources, and runtime binding are behind this support.

Language Preferences

The Language pane of the International system preferences (Figure 1) lets users set language based preferences for their computing environment. The preferences system (described in *Runtime Configuration Guidelines*) stores an ordered list of languages as a per-user default under the key `AppleLanguages`. Thus a user who understands more than one language can specify alternatives if an application does not include a localization for his primary language.

Figure 1 Language pane of the System Preferences International module



When a bundle requests a resource, the bundle interfaces try to choose a resource whose localization most closely matches the language and region settings in the user preferences. If a localization does not exist for the user's preferred language, the search continues for the user's second preferred language, and third preferred language, and so on. If none of the bundle's localizations match the user preferences, the interfaces return the localization used during development of the bundle.

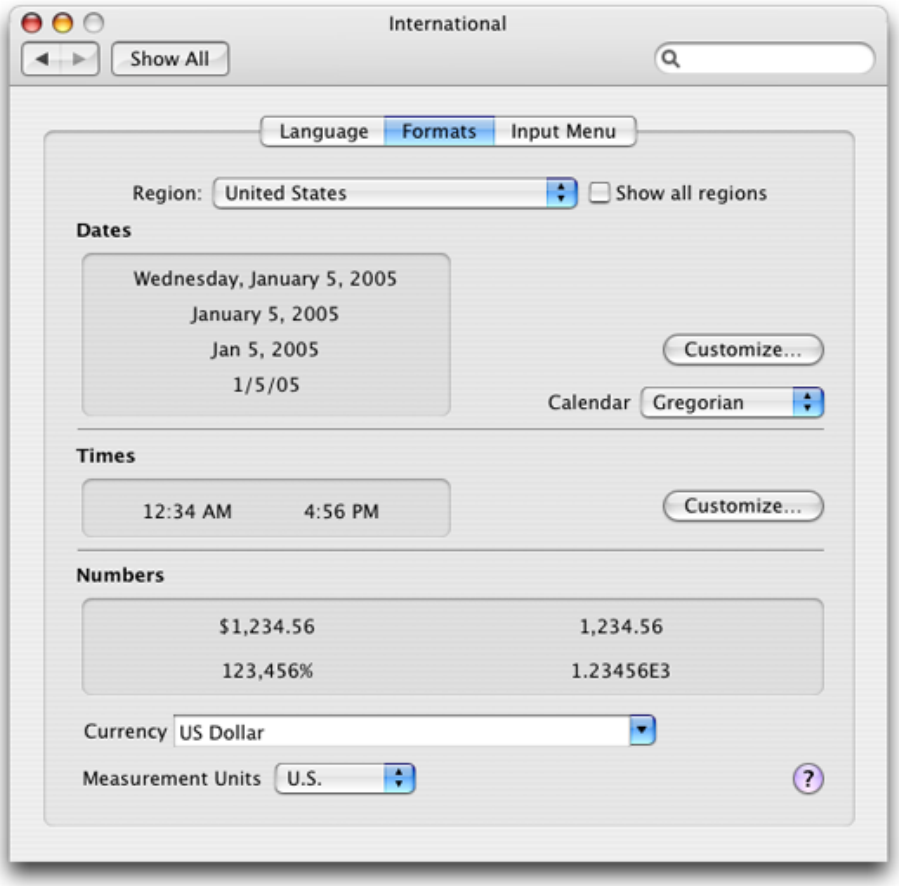
Note: For more information on how you store language-specific resources in a bundle, see *Bundle Programming Guide* in Core Foundation Documentation. For information on how to identify your bundle localizations to the CFBundle functions or `NSBundle` class, see "Specifying Supported Localizations in Your Bundle" (page 29).

Core Foundation Bundle Services (CFBundle) helps C-based applications retrieve the localized resources they need. The Cocoa `NSBundle` class provides the same capability for Objective-C and Java-based applications. In general, your application does not need to know which language is preferred by the user. The CFBundle and `NSBundle` interfaces automatically return the resource file that is most appropriate for the current user.

Locale Preferences

Locale preferences convey regional differences for the way dates, times, and numbers are displayed. Mac OS X includes predefined locales for many world regions. However, the user can also customize the information in these default locales to further personalize the system. Figure 2 shows the Formats pane of the International system preferences, where users can customize their locale settings.

Figure 2 Locale information in Formats pane



In most cases, your code should never have to worry about applying locale preferences. Most APIs take the user's preferences into account when getting or formatting locale-sensitive data. However, there may still be situations where you need to get locale information. For example, you might want to display the current locale settings, do comparisons of different locales, or apply locales other than the current locale to a specific piece of data. In those situations, the Core Foundation provides the CFLocale opaque type for getting locale information. For more information on locales and how to get locale-related data, see *Locales Programming Guide* in Core Foundation Internationalization Documentation.

Bundles

An **application bundle**, also known as a **file package** or **application wrapper**, is a directory containing an executable file and its associated resources. Application bundles are opaque directories, so to the user they look like files. Double-clicking an application bundle causes the application to be launched by the system.

Mac OS X supports other types of bundles, such as plug-ins and frameworks, too. A plug-in bundle contains code and resources like an application bundle but cannot be launched directly by the user. A framework bundle contains a dynamic shared library and resources to support that library. Unlike many other types of bundles, a framework is not stored in an opaque directory. The framework bundle directory is open so that developers can examine any header files and resources included with the framework.

Inside an application, the term "bundle" is often used to represent an instance of the `NSBundle` class or the `CFBundleRef` type. You use these objects to acquire resources from a bundle directory. Both objects automatically take the user's language preferences into account when looking for resources.

For more information about how bundles locate resources, see *Bundle Programming Guide*.

The Structure of a Mac OS X Application Bundle

An application bundle has a structure similar to the following:

Listing 1 Application Bundle Structure

```
MyApp.app/                          application wrapper
    Contents/
        MacOS/
            MyApp                   application executable
        Info.plist
        Resources/                  all resources go here
            MyApp.icns
            PeaceSign.tiff          non-localized resource
            en.lproj/               resources localized for English
                StopSign.eps        localized image
                Localizable.strings dynamically displayed text
                MyApp.nib           default platform nib
                MyApp Help/         help files
            fr.lproj/               resources localized for French
                StopSign.eps
                Localizable.strings
                MyApp.nib
                MyApp Help/         help files
            ja.lproj/               resources localized for Japanese
...
```

As you can see from this structure, your internationalized application needs one `.lproj` directory for each supported localization. The directories are named according to the language-version of the resources they contain. You do not have to create or populate these folders manually; Xcode does much of that work for you.

Each `.lproj` directory initially contains a copy of your application's native-language resource files. The resource file names must be identical in each of the `.lproj` directories. When you are ready to localize, you give the resource files to the translators to work on. They then translate the contents of the files and return them to you for inclusion in your Xcode project.

If your application displays resource file names to the user (such as image file or sound file names), you should store the resource file names themselves in a `.strings`file. This way, you can translate the name of the resource file as well as its contents. To load the resource, you first request the resource file name from the corresponding `.strings` file. Once you have the name, you can load the resource as usual.

Files within the `.lproj` directories can be of the following types:

- **Nib files** These files, created by the Interface Builder application, are encoded archives of user interface objects, custom-object references, and inter-object connections. The encoded user interface objects can include statically-displayed strings, images, and sounds.

  Nib files are typically localized directly, using Interface Builder. A localizer takes a nib file (or, more typically, your application's set of nib files), translates all the user interface strings, and makes other adjustments as necessary, such as resizing user interface objects and replacing culture-specific images and sounds.

  It's a good idea to design your application so that each window or panel is stored in its own nib file. This improves the performance of your application and also permits localization to progress incrementally.
- **Images** Cocoa applications can use `EPS`, `TIFF`, `PICT`, `GIF`, `JPEG`, `BMP`, and `ICO` files, as well as other supported image-file formats. You can also include QuickTime movies.
- **Localized text strings** A `.strings` file (`Localizable.strings` by default) contains dynamically-displayed strings for a given locale. For more information, see "Strings Files" (page 37).
- **Sounds**
- **Online help** Online help files are in `HTML`. The application's property list allows you to specify which file your application should open when the user chooses the Help command. A localized variant of your help should be placed in the appropriate `.lproj` directory for each targeted localization.

For localization, you'll send the source-language `.lproj` folder (`en.lproj` for developers in English-speaking countries) to a localization service, or to your in-house translation department. You'll also need to send the compiled application to allow your localizer to view dynamically-loaded resources in context, to ensure appropriate translations, and to ensure adequate dimensions for user interface elements that display dynamically-loaded resources. For each target language, they'll send back a `.lproj` folder in which each resource file has been appropriately localized.

The tool used to edit resources varies by resource type. You can edit `.strings` files with any text editor. If possible, use an editor that can save text in Unicode format, such as the TextEdit application included with Mac OS X. String files saved in Unicode format can be used in a running application directly, while other file formats require conversion first. Localizers edit nib files using Interface Builder and edit other resources (images, help files, sounds, and so on) with a tool appropriate to the file format and to your needs.

Language and Locale Designations

Mac OS X supports existing and forthcoming standards for the identification of languages and locales. All versions of Mac OS X support the International Organization for Standardization (ISO) naming conventions for identifying language and locale information. Specifically, Mac OS X supports the BCP 47 specification for identifying languages.

Important: If your software runs in versions of Mac OS X prior to version 10.4, you must continue to use the existing ISO language and locale ID conventions. Use of the tags found in the BCP 47 specification will not work on versions of Mac OS X prior to 10.4.

Using the available conventions, you can distinguish between different languages and between different regional dialects of a single language. The following sections show you how to specify this information in your code.

Language Designations

For language designations, Mac OS X supports both ISO 639-1 and ISO 639-2 conventions. The ISO 639-1 specification uses a two-letter code to identify a language and is the preferred way to identify languages in Mac OS X. However, if an ISO 639-1 code is not available for a particular language, you may also use the three-letter designators defined by the ISO 639-2 specification. Table 1 lists ISO designators for a subset of languages. Note that there is no ISO 639-1 designator for Hawaiian and so you must use the ISO 639-2 designator.

Table 1 Examples of ISO language designations

Language	ISO 639-1	ISO 639-2
English	`en`	`eng`
French	`fr`	`fre`
German	`de`	`ger`
Japanese	`ja`	`jpn`
Hawaiian	no designator	`haw`

Note: For a complete list of ISO 639-1 and ISO 639-2 codes, go to http://www.loc.gov/standards/iso639-2/php/English_list.php.

Regional Designations

For regional designations, Mac OS X supports the ISO 3166-1 conventions. This specification uses a two-letter, capitalized code to identify a specific country. By catenating a language designator with an underscore character and a regional designator, you get a designator that identifies the locale for a specific language and country. Table 2 lists the locale designators for a subset of languages and countries.

Table 2 Examples of ISO regional designators

Regional dialect	Region Designator
English (United States)	`US`
English (Great Britain)	`GB`
English (Australian)	`AU`
French (France)	`FR`
French (Canadian)	`CA`

Note: For a complete list of ISO 3166-1 codes, go to http://www.iso.ch.

Language and Locale IDs

A **language ID** designates a written language (or orthography) and can reflect either the generic language or a specific dialect of that language. To specify a language ID, you use a language designator by itself. To specify a specific dialect of a language, you use a hyphen to combine a language designator with a region designator. Thus, the English language as it is spoken in Great Britain would yield a language ID of `en-GB`, while the English language spoken in the United States would have a language ID of `en-US`.

A **locale ID** identifies a specific location where a given language is spoken. To specify a locale ID, use an underscore character to combine a language designator with a region designator. The locale ID for English-language speakers in Great Britain is `en_GB`, while the locale for English-speaking residents of the United States is `en_US`. Although locale IDs and language IDs might seem nearly identical, there is a subtle difference. A language ID identifies a written and spoken language only. A locale identifies a region and its conventions and has a more cultural context.

To illustrate the difference between language IDs and locale IDs, consider the following example. The dialect for a resident of Great Britain is specified by the code `en-GB`. The commonly used locale for that same person is `en_GB`. If you wanted to be very precise when specifying the locale, you could specify the locale code as

en-GB_GB. This specifies a person who speaks the British dialect of English and who resides in Great Britain. If that same person moved to the United States, the appropriate locale would be en-GB_US, which would identify a person who speaks British English but uses the regional settings associated with the United States.

Mac OS X v10.4 and later supports the language ID tags defined in the BCP 47 specification. In addition to the ISO 3166-1 region codes, the draft of this standard (available at http://www.rfc-editor.org/) adds support for tags ranging in length from 3 to 8 characters. The use of these tags makes it possible to separate dialect or script information from a specific region or country.

Particularly in Chinese dialects, a region code is not always the best way to specify the proper dialect or script. For example, traditional Chinese (Han) is the default language spoken in Taiwan and is identified by the code zh_TW in Mac OS X v10.3.9 and earlier. However, traditional Chinese is also commonly spoken in Hong Kong and Macao, which means the zh_TW designator is not entirely accurate in those locations. The new standard defines new tags for the traditional Chinese (Hant) and simplified Chinese (Hans) scripts. Thus, traditional Chinese spoken in any country uses the code zh-Hant. Traditional Chinese, as it is spoken in Taiwan, now uses the locale code zh-Hant_TW.

Table 3 lists some of the other custom tags that identify a particular dialect or script.

Table 3 Custom language ID tags

Language ID	Description
az-Arab	Azerbaijani in the Arabic script.
az-Cyrl	Azerbaijani in the Cyrillic script.
az-Latn	Azerbaijani in the Latin script.
sr-Cyrl	Serbian in the Cyrillic script.
sr-Latn	Serbian in the Latin script.
uz-Cyrl	Uzbek in the Cyrillic script.
uz-Latn	Uzbek in the Latin script.
zh-Hans	Chinese in the simplified script.
zh-Hant	Chinese in the traditional script.

Important: In Mac OS X v10.4 and later, you can use the new `Hant` and `Hans` tags instead of the older `zh_TW` and `zh_CN` tags for `.lproj` directory names if you choose. You must not use these tags (or any of the newer tags) in Mac OS X v10.3.9 and earlier, however. For these older applications, you should use the Core Foundation and Cocoa routines to obtain the canonical form of a given language or locale tag before using that tag in your code. For information on how to get the canonical tags, see "Getting Canonical Language and Locale IDs" (page 30).

Language-Specific Project Directories

The more general you make your localized resources, the more regions you can support with a single set of resources. This can save a lot of space in your bundle and helps reduce translation costs. For example, if you did not need to distinguish between different regions of the English language, you could include a single `en.lproj` directory to support users in the United States, Great Britain, and Australia. Of course, the decision to use common resources over region-specific versions depends entirely on your product and the needs of your users.

Important: Even if you support region-specific localizations, you should always provide a complete set of common resources that are not region-specific.

When searching for resources, the system bundle routines try to find the best match between the `.lproj` directories in your bundle and the user's language and region preferences. The bundle routines look for the requested resource in region-specific directories first, followed by the more generalized language directory. Thus, if you had localizations for United States, Great Britain, and Australian users, the bundle routines would search the appropriate region directory (`en_US.lproj`, `en_GB.lproj`, or `en_AU.lproj`) first, followed by the `en.lproj` directory. For more information about how bundles search for resources, see "Searching for Bundle Resources" in *Bundle Programming Guide*.

Getting Language Names from Designators

Prior to Mac OS X v10.4, the ISO language and region designators are the recommended way of identifying languages in the system. However, few users can recognize languages by their ISO designators. If you need to display the actual name of a language to a user, you can use the Carbon functions defined in `MacLocales.h` to convert the designators into localized language names. For more information, see *Locale Utilities Reference*.

If your software runs in Mac OS X v10.4 and later, you should not use the functions in `MacLocales.h`. Instead, use the `CFLocaleCopyDisplayNameForPropertyValue` function to get the correct display name for the language or locale ID.

Using Custom Designators

It is possible to use a language or locale abbreviation that is not known to the CFBundle functions or `NSBundle` class. For example, you could create your own language designations for a language that is not yet listed in the ISO conventions. Use of custom designators is discouraged, however.

If you choose to create a new designator, be sure to follow the rules found in sections 2.2.1 and 4.5 of BCP 47. Tags that do not follow these conventions are not guaranteed to work. When using custom tags, you must ensure that the abbreviation stored by the user's language preferences matches the designator used by your `.lproj` directory exactly.

Legacy Language Designators

In addition to the ISO language designators, previous versions of Mac OS X also supported a set of legacy designators. These designators let you specify a language by name, instead of by a two or three character code. Designators included names such as `English`, `French`, `German`, `Japanese`, `Chinese`, `Spanish`, `Italian`, `Swedish`, and `Portuguese` among others. Although these names are still recognized and processed by the CFBundle functions or `NSBundle` class, their use is deprecated and support for them in future versions of Mac OS X is not guaranteed. Use the codes described in "Language Designations" (page 19) and "Regional Designations" (page 20) instead.

Guidelines for Internationalization

The following sections provide recommendations on how to internationalize your software to best take advantage of Mac OS X.

Use Canonical Language and Locale IDs

Mac OS X 10.3 added support for obtaining canonical locale IDs through the functions of CFLocale. In Mac OS X v10.4 and later, support was added for getting canonical language IDs. (Cocoa support was also added in Mac OS X v10.4.) If you get the user's preferred locale ID through the CFBundle functions or `NSBundle` class, then you should already be receiving a canonical locale ID.

For information about language and locale IDs, see "Language and Locale Designations" (page 19). For information on how to get canonical language and locale IDs, see "Getting Canonical Language and Locale IDs" (page 30).

Use Bundles

The Mac OS X bundle mechanism simplifies the process of localization by letting you place localized resources in separate directories named for the language or region they support. Both the CFBundle functions and `NSBundle` class use this collection of resource folders to localize a program based on the current user's language and region settings. This mechanism makes it possible for a bundled program to display itself in different languages for different users. Organizing resources by directory also makes it easy to add new localizations later.

Support Unicode Text

Applications should use Unicode-based technologies in all situations that involve displaying text to the user. Unicode support is essential for multibyte languages and Mac OS X supports some languages (including Arabic, Thai, Greek, and Hawaiian) only through Unicode. Regardless of whether you localize your software for multibyte languages, you should still support Unicode so that users can enter text data in their native language.

Mac OS X provides extensive support for managing and displaying Unicode text. Technologies such as ATSUI, MLTE, Quartz, and Cocoa Text all leverage Unicode in the display of text. Core Foundation and Cocoa strings also provide a way to store Unicode strings in your application. Use of these technologies makes it possible to support virtually any language.

While Unicode is important for text that is displayed to the user, you do not need to use it everywhere in your application. For example, you might not need to use Unicode for debugging text or internal logging mechanisms, unless you want those messages to be translated and displayed to users.

Guidelines for Adding MultiScript Support

An application that provides multiscript support can accurately display text in various scripts simultaneously. Such an application can accept text input, display text, and print text containing the scripts of different languages in the same document, regardless of a user's language preferences. If an application were not prepared to offer multiscript support, some of this text would not appear correctly.

Multiscript support is becoming an increasingly important and expected feature not only for operating systems but for third-party applications. With an internationalized operating system such as Mac OS X, some users expect to be able to create a document in one language, change their language preference, and then open the document as they last saved it.

To add multiscript support to an application on Mac OS X, you must use the appropriate Unicode technologies and API. The text classes in the Cocoa framework automatically provide multiscript support. For Carbon and other non-Cocoa applications, you should use the following technologies:

- Wherever possible, use CFString from Core Foundation String Services instead of C or Pascal strings.

  CFString objects internally store and handle Unicode data without requiring you to have any specific knowledge of the global character-set standard. If you need to convert between Unicode and C and Pascal strings in other encodings, use the facilities that CFString provides. However, avoid converting CFString objects to and from C strings or Pascal strings if possible. Such conversions are computationally expensive and frequently introduce bugs affecting multiscript presentation. If you cannot find any CFString or Unicode-aware API to use in a certain situation, convert the string to the application's text encoding.
- To format dates use the CFDateFormatter interface in Core Foundation.
- To format numbers, use the CFNumberFormatter interface in Core Foundation.
- For localized strings, use Core Foundation's `CFCopyLocalizedString` (and related macros) instead of `GetIndString`.

  Strings returned by `GetIndString` are specific to the current script system and thus cannot represent multiscript text. `CFCopyLocalizedString` returns Unicode-based CFString objects, which can represent multiscript text. Generally, your code should use dynamic text processing over static text in your code. See "Loading String Resources Into Your Code" (page 41) for more information.
- Avoid directly accessing system layers below Core Services.

  You should be able to obtain most of the functionality available in the kernel environment (particularly BSD and Mach calls) using the Core Services frameworks. As much as possible, avoid calling BSD functions directly. For accessing the file system, use Core Foundation URL Services (CFURL), the File Manager data type `FSRef`, or the `NSFileManager` object in Cocoa.
- Avoid using the TextEdit API.

  The TextEdit API is capable of dealing with multiscript text. However, it requires you to manage script fonts and style runs yourself. The Multilingual Text Engine (MLTE) provides a much simpler API to handle multiscript text based on Unicode.
- Never assume text data to be in the MacRoman encoding.

You can no longer assume that all text data uses MacRoman encoding or ignore text encoding issues altogether. Untagged text data unaccompanied by script code is not necessarily in the system script (Roman). If you make this assumption, users may be presented with incoherent text. In the worst case, your text processing routines could misinterpret the text and either corrupt user data or crash the system.

For more information on file encodings in Mac OS X, see "File Encodings and Fonts" (page 55).

Carefully Consider Translatable Strings

When writing text for your application, try to think about how that text might be translated. Grammatical differences between languages can make some text difficult to translate, even with the formatting options available in strings files. In particular, strings that require differentiation between singular and plural forms (because of a count variable, for example) may require different strings in different circumstances. Talk to your translation team about ways to eliminate potential translation problems before they occur.

For more information about strings and strings files, see "Strings Files" (page 37)

Getting the Current Language and Locale

For most developers, choosing a localization to use is not something you ever have to do. The `NSBundle` and CFBundle interfaces automatically apply the user's preferences to determine which localized resource files to return in response to a programmatic request. However, there may be situations beyond simply loading resources from a bundle that would require your program to know which localization was in use. For those situations, both `NSBundle` and CFBundle provides ways for you to get that information.

Specifying Supported Localizations in Your Bundle

Before discussing the techniques of how to get localizations, it is important to remember how a bundle communicates its supported localizations to the system. Most bundled applications contain a `Resources` directory, inside of which reside language-specific project (`.lproj`) directories. Each of these `.lproj` directories contains the resources associated with a specific language or regional dialect. `NSBundle` and CFBundle look for these directories and use them to build a list of supported localizations. However, this is not the only way to build this list.

An application can notify the system that it supports additional localizations through its information property list (`Info.plist`) file. To specify localizations not included in your bundle's `.lproj` directories, add the CFBundleLocalizations key to this file. The value for the key is an array of strings, each of which contains an ISO language designator as described in "Language and Locale Designations" (page 19).

Getting the Preferred Localizations

If you have a Carbon application or are using the CFBundle functions to manage your program bundle, you can use the `CFBundleCopyPreferredLocalizationsFromArray` function to get the most relevant localizations. When calling this method, you must pass in a list of localizations your software supports. You can use the function `CFBundleCopyBundleLocalizations` to generate this list for you using the bundle information.

This function compares the supported localizations against the user's language and region preferences. From this comparison, it returns an array of strings, one of which is the language-only localization most appropriate for the user. If a region-specific localization is also available, it returns that localization as well, giving it preference over the language-only localization.

If you are writing a Cocoa application, you can use the `preferredLocalizationsFromArray:` and `localizations` methods of `NSBundle` to implement the same behavior as `CFBundleCopyPreferredLocalizationsFromArray` and `CFBundleCopyBundleLocalizations`. You can also use the method `preferredLocalizations` as a shortcut to perform both actions with a single method. As with the CFBundle functions, the `preferredLocalizations` and `preferredLocalizationsFromArray:` methods return an array of strings containing the language designators in use by the bundle.

Mac OS X stores each user's list of preferred languages in that user's defaults database, along with other system-wide and application-specific preferences. You can access this database using the preference-management routines found in both Core Foundation or Cocoa. An array of preferred languages is associated with the `AppleLanguages` key. The currently selected locale is associated with the `AppleLocale` key. Both of these keys are in the `NSGlobalDomain`.

Important: It is recommended you use the CFBundle functions or `NSBundle` methods to read the preferred locale and languages instead of reading the contents of the defaults database directly. The codes found in the database may not include the canonical forms of the language or locale IDs.

Getting Canonical Language and Locale IDs

Prior to Mac OS X v10.4, language dialects were specified by combining a language designator with a region designator. With the introduction of support for custom dialect codes (see "Language and Locale IDs" (page 20)), getting the appropriate language code is now somewhat more complicated. Fortunately, Mac OS X provides routines to help you determine the appropriate language and locale codes based on the information you have.

In Mac OS X v10.3, the CFLocale opaque type was introduced in Core Foundation. One of the functions introduced with this type is the `CFLocaleCreateCanonicalLocaleIdentifierFromString` function, which takes the locale code you specify and returns an appropriate canonical version. This function is particularly useful for converting older locale strings, such as the older, English-based .lproj directory names, into the ISO-compliant names.

In Mac OS X v10.4, the `CFLocaleCreateCanonicalLanguageIdentifierFromString` function was added to perform the same canonical conversion for language and dialect codes. For example, this function converts the old specifier for traditional Chinese (`zh_TW`) to the more modern version (`zh-Hant`). Also in Mac OS X v10.4, Cocoa added the `NSLocale` class to provide Objective-C wrappers for the corresponding Core Foundation functions.

If you use CFBundle or `NSBundle` to retrieve language-specific resources from your bundle, then you do not need to worry about language identifiers directly. The CFBundle and `NSBundle` routines automatically handle language and locale IDs in canonical and non-canonical forms.

If your code requires Mac OS X v10.4 or later, you should start using the new canonical forms for language and locale IDs. Some older language codes are replaced by newer codes in v10.4. In addition to several of the Chinese language codes, support for the newer Norwegian ISO code (`nb`) is now available and should be preferred over the older version.

Note: If your program also supports versions of Mac OS X prior to 10.3, you may need to maintain your own table of canonical IDs.

Getting Language and Locale Preferences Directly

There may be situations where you want to get the preferred locale ID or the list of languages directly from the user preferences. Mac OS X stores each user's list of preferred languages in that user's defaults database. The list of preferred languages is identified by the defaults key `AppleLanguages` and is stored in the global variable `NSGlobalDomain`. You can access that list using the `NSUserDefaults` class in Cocoa or the Core Foundation preferences functions.

Important: If you get the user language preference from the defaults database, you must get the canonical form using the `CFLocaleCreateCanonicalLanguageIdentifierFromString` function (in Mac OS X v10.4 and later) or `CFLocaleCreateCanonicalLocaleIdentifierFromString` function (in Mac OS X v10.3 and later) before using the identifier.

The following example shows you to get the list of preferred languages from the defaults database using Cocoa. The returned array contains the languages associated with the `AppleLanguages` key in the user's preferred order. Thus, in most cases, you would simply want to get the first object in the array.

```
NSUserDefaults* defs = [NSUserDefaults standardUserDefaults];
NSArray* languages = [defs objectForKey:@"AppleLanguages"];
NSString* preferredLang = [languages objectAtIndex:0];
```

The locale for the current user is also stored in the defaults database under the `AppleLocale` key.

Important: Although you can get the user's preferred settings from the defaults database, it is recommended you use the CFBundle functions or `NSBundle` class instead. The associated functions and methods of those objects return the preferred language or locale that is also supported by your application. (Bear in mind that the returned values may not correspond directly with the user's exact preferences.)

Preparing Your Nib Files for Localization

The native integrated development environment (IDE) for Mac OS X consists of Xcode, Interface Builder, and a suite of build, debugging, and performance tools. Developers use Interface Builder to create user interfaces for their applications. Interface Builder saves these interfaces as XML archives called nib files. You can localize nib files just as you can localize image and sound files.

Nib files store the user interface of your application, including windows, dialogs, and user-interface elements such as buttons, sliders, text objects, and help tags for these elements. A nib file also holds the connections between these objects that cause actions to be performed when the user activates controls.

Note: This section talks about Interface Builder and its nib files. However, much of the discussion is applicable to localized user-interface archives created by other tools.

Using Nib Files Effectively

Translators typically localize the pieces of the user interface all at once, adjusting graphic elements to account for changes in string lengths. In any medium-size or large application, it's usually a good idea to put each window or panel (that is, dialog) in its own nib file. This practice not only makes it possible to load the user interface lazily (that is, to load pieces as they're used), but it also permits localization to progress in more incremental steps. It's also a good idea to put the menus of the application in a separate nib file.

Translators can use a combination of Interface Builder, AppleGlot, and `ibtool` to localize nib files. Using these tools, the translator would open the nib files in a given *language*`.lproj`directory, localize the strings, change the sizes of the user-interface elements to accommodate the new strings, and save the changes back to the nib files. There are a few other things to watch out for:

- Objects in a nib file typically have connections between them that should not be broken. Make sure you lock all connections (an option in Interface Builder preferences) before handing your nibs off to translation.
- Dialogs and windows usually have minimum or maximum sizes that are specified through the inspector. If a dialog or window must be made wider for a given language, the translator should check to make sure that the minimum size is also updated to an appropriate value.
- Some user-interface objects support help tags—explanatory text that appears when the user moves the mouse over the object for a short period. You can define the help tags for an object in Interface Builder's inspector, where the translator can also localize those strings.

In your nib files, it is also important to remember that localization will likely change the size of visible text in your windows. If text labels do not have room to expand, the localizers may have to adjust the size of your window or the positions of the label or other controls. In general, you should expect text labels in English to expand by up to 40% in length during translation.

Using ibtool

It can be tedious to translate nib files manually and even more tedious to propagate subsequent design changes to the already localized nib files in your project. Fortunately, the `ibtool` command-line utility makes propagating changes to other nib files much easier. This tool operates on a dictionary of all the strings in your nib file. Using it, you can extract the strings from the nib file, translate them, and then inject them back into the nib file. Alternatively, if you already have valid translations, you can inject the already-translated strings into a master nib file containing any recent design changes to update the localized nib files.

Note: The `ibtool` command-line utility replaces the `nibtool` utility that was present prior to Mac OS X v10.5.

The following example shows you how to extract the English strings from the main nib of a project into a new strings file. You can store a copy of the resulting file for use later or give the file to the translation team.

```
ibtool --generate-strings-file MainMenu.strings en.lproj/MainMenu.nib
```

The following example takes the original English-based nib file and merges it with the translated strings to create a localized version of the nib:

```
// The translated strings are in the de.lproj directory
//  with the same name as the original file.
ibtool --strings-file de.lproj/MainMenu.strings --write de.lproj/MainMenu.nib
en.lproj/MainMenu.nib
```

If you already have a localized nib file and want to update it incrementally, `ibtool` provides several options. You can use the -I option to inject a new set of translations into the existing nib file. For more information see the `ibtool` man page.

Important: Remember that translating the strings in your nib file is only one step of the localization process. The localizer may need to adjust the layout of views and controls to account for changes in string length. Also, `ibtool` does not automatically update date and time formats associated with formatter objects.

Generating Localized Nib Files

Internationalizing applications for Mac OS X involves, in part, putting the resources localized for a particular language or region in the proper bundle location. In some situations, you might have to do this task by hand, such as with Code Fragment Manager (CFM) applications. Fortunately, for most occasions there are tools to help you.

Xcode provides the File Reference Inspector to assist you with internationalization. To internationalize a resource file in Xcode 3.1, do the following:

1. Add the resource file to Xcode.
    a. Expand your project in the Groups & Files pane.
    b. Select the Resources folder.

   c. Choose Project > Add to Project….

   d. Use the file browser to navigate to the resource, select it, and click Open.

   e. In the dialog that Xcode displays, select "Copy items into destination group's folder", choose one of the "relative" reference styles (for example, "Relative to Project"), and select the desired target. Click Add.

2. Select the resource file and open the Inspector window (File > Get Info).

3. If the file is not already localizable, click the Make File Localizable button in the General pane of the Inspector window.

4. In the General pane again, click the Add Localization button. In the sheet that appears, select or type the locale you want to add. Xcode copies the resource to the other `.lproj` directory.

If your resource file is a plain text file, you should be sure to set the File Encoding for each localized copy. You can set the default localization of the file before you make the file localizable. You can use different encodings for different localized versions of the file. To set the encoding for a specific localization, choose that localization in the Groups and Files pane and change the setting in the Inspector window.

Strings Files

An important part of the localization process is to localize all of the text strings displayed by your application. By their nature, strings located in nib files can be readily localized along with the rest of the nib file contents. Strings embedded in your code, however, must be extracted, localized, and then reinserted back into your code. To make the maintenance of your code easier, Mac OS X ships with a tool that extracts strings from your code and puts them into resource files where they can be localized more easily. This article talks about the process for extracting strings from your code and managing the corresponding resources files.

About Strings Files

Resource files that contain localizable strings are referred to as **strings files** (with the deliberate extra 's' in the word "strings") because of their filename extension, which is `.strings`. You can create strings files manually or programmatically depending on your needs. The standard strings file format consists of one or more key-value pairs along with optional comments. The key and value in a given pair are strings of text enclosed in double quotation marks, separated by an equal sign, and terminated by a semicolon. (You can also use a property list format for strings files. In such a case, the top-level node is a dictionary and each key-value pair of that dictionary is a string entry.) Although the inclusion of comments is optional, they do provide a useful way to communicate contextual information to the translator about how each string is used. Listing 1 shows a simple strings file with two non-localized entries for the default language.

Listing 1 A simple strings file

```
/* Insert Element menu item */
"Insert Element" = "Insert Element";
/* Error string used for unknown error types. */
"ErrorString_1" = "An unknown error occurred.";
```

A typical application has at least one strings file per localization, that is, one strings file in each of the bundle's `.lproj` subdirectories. The name of the default strings file is `Localizable.strings` but you can create strings files with any file name you choose. Creating strings files is discussed in more depth in "Creating Strings Resource Files" (page 38).

Note: It is recommended that you save strings files using the UTF-16 encoding, which is the default encoding for standard strings files. It is possible to create strings files using other property-list formats, including binary property-list formats and XML formats that use the UTF-8 encoding, but doing so is not recommended. For more information about the standard strings file format, see "Creating Strings Resource Files" (page 38). For more information about Unicode and its text encodings, go to http://www.unicode.org/ or http://en.wikipedia.org/wiki/Unicode.

Creating Strings Resource Files

Although you can create strings files manually, it is rarely necessary to do so. The easiest way to create strings files is to write your code using the appropriate string-loading macros and then use the `genstrings` command-line tool to extract those strings and create strings files for you.

The following sections describe the process of how to set up your source files to facilitate the use of the `genstrings` tool. For detailed information about the tool, see `genstrings` man page.

Choosing Which Strings to Localize

When it comes to localizing your application's interface, it is not always appropriate to localize every string used by your application. Translation is a costly process, and translating strings that are never seen by the user is a waste of time and money. Strings that are not displayed to the user, such as notification names used internally by your application, do not need to be translated. Consider the following example:

```
if (CFStringHasPrefix(value, CFSTR("-")) {    CFArrayAppendValue(myArray,
value);};
```

In this example, the string "`-`" is used internally and is never seen by the user; therefore, it does not need to be placed in a strings file.

The following code shows another example of a string the user would not see. The string `"%d %d %s"` does not need to be localized, since the user never sees it and it has no effect on anything that the user does see.

```
matches = sscanf(s, "%d %d %s", &first, &last, &other);
```

Because nib files are localized separately, you do not need to include strings that are already located inside of a nib file. Some of the strings you should localize, however, include the following:

- Strings that are programmatically added to a window, panel, view, or control and subsequently displayed to the user. This includes strings you pass into standard routines, such as those that display alert boxes.
- Menu item title strings if those strings are added programmatically. For example, if you use custom strings for the Undo menu item, those strings should be in a strings file.
- Error messages that are displayed to the user.
- Any boilerplate text that is displayed to the user.
- Some strings from your application's information property list (`Info.plist`) file; see *Runtime Configuration Guidelines*.
- New file and document names.

About the String-Loading Macros

The Foundation and Core Foundation frameworks define the following macros to make loading strings from a strings file easier:

- Core Foundation macros:
  - `CFCopyLocalizedString`
  - `CFCopyLocalizedStringFromTable`
  - `CFCopyLocalizedStringFromTableInBundle`
  - `CFCopyLocalizedStringWithDefaultValue`
- Foundation macros:
  - `NSLocalizedString`
  - `NSLocalizedStringFromTable`
  - `NSLocalizedStringFromTableInBundle`
  - `NSLocalizedStringWithDefaultValue`

You use these macros in your source code to load strings from one of your application's strings files. The macros take the user's current language preferences into account when retrieving the actual string value. In addition, the `genstrings` tool searches for these macros and uses the information they contain to build the initial set of strings files for your application.

For detailed information about how to use these macros, see "Loading String Resources Into Your Code" (page 41).

Using the Genstrings Tool to Create Strings Files

At some point during your development, you need to create the strings files needed by your code. If you wrote your code using the Core Foundation and Foundation macros, the simplest way to create your strings files is using the `genstrings` command-line tool. You can use this tool to generate a new set of strings files or update a set of existing files based on your source code.

To use the `genstrings` tool, you typically provide at least two arguments:

- A list of source files
- An optional output directory

The `genstrings` tool can parse C, Objective-C, and Java code files with the `.c`, `.m`, or `.java` filename extensions. Although not strictly required, specifying an output directory is recommended and is where `genstrings` places the resulting strings files. In most cases, you would want to specify the directory containing the project resources for your development language.

The following example shows a simple command for running the `genstrings` tool. This command causes the tool to parse all Objective-C source files in the current directory and put the resulting strings files in the `en.lproj` subdirectory, which must already exist.

```
genstrings -o en.lproj *.m
```

The first time you run the `genstrings` tool, it creates a set of new strings files for you. Subsequent runs replace the contents of those strings files with the current string entries found in your source code. For subsequent runs, it is a good idea to save a copy of your current strings files before running `genstrings`. You can then diff the new and old versions to determine which strings were added to (or changed in) your project. You can then use this information to update any already localized versions of your strings files, rather than replacing those files and localizing them again.

Within a single strings file, each key must be unique. Fortunately, the `genstrings` tool is smart enough to coalesce any duplicate entries it finds. When it discovers a key string used more than once in a single strings file, the tool merges the comments from the individual entries into one comment string and generates a warning. (You can suppress the duplicate entries warning with the `-q` option.) If the same key string is assigned to strings in different strings files, no warning is generated.

For more information about using the `genstrings` tool, see the `genstrings` man page.

Creating Strings Files Manually

Although the `genstrings` tool is the most convenient way to create strings files, you can also create them manually. To create a strings file manually, create a new file in TextEdit (or your preferred text-editing application) and save it using the Unicode UTF-16 encoding. (When saving files, TextEdit usually chooses an appropriate encoding by default. You can force the encoding of a plain text file by choosing Save As and selecting the appropriate encoding. If the file is not a plain text file, you must first convert it by choosing Format > Make Plain Text.) The contents of this file consists of a set of key-value pairs along with optional comments describing the purpose of each key-value pair. Key and value strings are separated by an equal sign, and the entire entry must be terminated with a semicolon character. By convention, comments are enclosed inside C-style comment delimiters (`/*` and `*/`) and are placed immediately before the entry they describe.

Listing 2 shows the basic format of a strings file. The entries in this example come from the English version of the `Localizable.strings` file from the TextEdit application. The left side of each equal sign represents the key, and the right side represents the value. A common convention when developing applications is to use a key name that equals the value in the language used to develop the application. Therefore, because TextEdit was developed using the English language, the English version of the `Localizable.strings` file has keys and values that match.

Listing 2 Strings localized for English

```
/* Menu item to make the current document plain text */
"Make Plain Text" = "Make Plain Text";
/* Menu item to make the current document rich text */
"Make Rich Text" = "Make Rich Text";
```

Listing 3 shows the German translation of the same entries. These entries also live inside a file called `Localizable.strings`, but this version of the file is located in the German language project directory of the TextEdit application. Notice that the keys are still in English, but the values assigned to those keys are in German. This is because the key strings are never seen by end users. They are used by the code to retrieve the corresponding value string, which in this case is in German.

Listing 3 Strings localized for German

```
/* Menu item to make the current document plain text */
```

```
"Make Plain Text" = "In reinen Text umwandeln";
/* Menu item to make the current document rich text */
"Make Rich Text" = "In formatierten Text umwandeln";
```

Detecting Nonlocalizable Strings

AppKit–based applications can take advantage of built-in support to detect strings that do not need to be localized and those that need to be localized but currently are not. You enable this support using arguments that you pass to your executable at launch time. If you are using Xcode, you set these arguments using the inspector for the corresponding executable. Open an inspector window for the desired executable (located in the Executables section of your project window) and select the Arguments tab. Add an entry for the new argument and include both the argument string and the desired value (separated by a space) in the entry. The available argument strings are as follows:

- The `NSShowNonLocalizableStrings` setting identifies strings that are not localizable. The strings are logged to the shell in upper case. This option occasionally generates some false positives but is still useful overall.
- The `NSShowNonLocalizedStrings` setting locates strings that were meant to be localized but could not be found in the application's existing strings files. You can use this setting to catch problems with out-of-date localizations.

You can also pass arguments to an application by running that application from the command line. For example, to use the `NSShowNonLocalizedStrings` setting with the TextEdit application, you would enter the following in Terminal:

```
/Applications/TextEdit.app/Contents/MacOS/TextEdit -NSShowNonLocalizedStrings
YES
```

Loading String Resources Into Your Code

The Core Foundation and Foundation frameworks provide macros for retrieving strings stored in strings files. Although the main purpose of these macros is to load strings at runtime, they also serve a secondary purpose by acting as markers that the `genstrings` tool can use to locate your application's string resources. It is this second purpose that explains why many of the macros let you specify much more information than would normally be required for loading a string. The `genstrings` tool uses the information you provide to create or update your application's strings files automatically. Table 1 lists the types of information you can specify for these routines and describes how that information is used by the `genstrings` tool.

Table 1 Common parameters found in string-loading routines

Parameter	Description
Key	The string used to look up the corresponding value. This string must not contain any characters from the extended ASCII character set, which includes accented versions of ASCII characters. If you want the initial value string to contain extended ASCII characters, use a routine that lets you specify a default value parameter. (For information about the extended ASCII character set, see the corresponding Wikipedia entry.)

Parameter	Description
Table name	The name of the strings file in which the specified key is located. The `genstrings` tool interprets this parameter as the name of the strings file in which the string should be placed. If no table name is provided, the string is placed in the default `Localizable.strings` file. (When specifying a value for this parameter, include the filename without the `.strings` extension.)
Default value	The default value to associate with a given key. If no default value is specified, the `genstrings` tool uses the key string as the initial value. Default value strings may contain extended ASCII characters.
Comment	Translation comments to include with the string. You can use comments to provide clues to the translation team about how a given string is used. The `genstrings` tool puts these comments in the strings file and encloses them in C-style comment delimiters (`/*` and `*/`) immediately above the associated entry.
Bundle	An `NSBundle` object or `CFBundleRef` type corresponding to the bundle containing the strings file. You can use this to load strings from bundles other than your application's main bundle. For example, you might use this to load localized strings from a framework or plug-in.

When you request a string from a strings file, the Core Foundation and Cocoa macros automatically use the available localizations in your application and the current user's language preferences to return the most appropriate string for that user. The string-loading macros search the application bundle for a strings file with the localization closest to the user's preferred language and return that string for your code to use. As long as your code is properly internationalized, you should not have to do anything else. For information about how an appropriate localization is chosen, see "Language Preferences" (page 13).

Using the Core Foundation Framework

The Core Foundation framework defines a single function and several macros for loading localized strings from your application bundle. The `CFBundleCopyLocalizedString` function provides the basic implementation for retrieving the strings. However, it is recommended that you use the following macros instead:

- `CFCopyLocalizedString(key, comment)`
- `CFCopyLocalizedStringFromTable(key, tableName, comment)`
- `CFCopyLocalizedStringFromTableInBundle(key, tableName, bundle, comment)`
- `CFCopyLocalizedStringWithDefaultValue(key, tableName, bundle, value, comment)`

There are several reasons to use the macros instead of the `CFBundleCopyLocalizedString` function. First, the macros are easier to use for certain common cases. Second, the macros let you associate a comment string with the string entry. Third, the macros are recognized by the `genstrings` tool but the `CFBundleCopyLocalizedString` function is not.

For additional information on how to use these macros, see Working With Localized Strings in *Bundle Programming Guide*. For macro and function syntax, see *CFBundle Reference*.

Using the Foundation Framework

The Foundation framework defines a single method and several macros for loading string resources. The `localizedStringForKey:value:table:` method of the `NSBundle` class loads the specified string resource from a strings file residing in the current bundle. Cocoa also defines the following macros for getting localized strings:

- `NSLocalizedString(key, comment)`
- `NSLocalizedStringFromTable(key, tableName, comment)`
- `NSLocalizedStringFromTableInBundle(key, tableName, bundle, comment)`
- `NSLocalizedStringWithDefaultValue(key, tableName, bundle, value, comment)`

As with Core Foundation, Apple recommends that you use the Cocoa convenience macros for loading strings. The main advantage to these macros is that they can be parsed by the `genstrings` tool and used to create your application's strings files. They are also simpler to use and let you associate translation comments with each entry.

For information about the syntax of the preceding macros, see *Foundation Functions Reference*. Additional methods for loading strings are also defined in *NSBundle Class Reference*.

Examples of Getting Strings

The following examples demonstrate the basic techniques for using the Foundation and Core Foundation macros to retrieve strings. Each example assumes that the current bundle contains a strings file with the name `Custom.strings` that has been translated into French. This translated file includes the following strings:

```
/* A comment */
"Yes" = "Oui";
"The same text in English" = "Le même texte en anglais";
```

Using the Foundation framework, you can get the value of the "`Yes`" string using the `NSLocalizedStringFromTable` macro, as shown in the following example:

```
NSString* theString;
theString = NSLocalizedStringFromTable (@"Yes", @"Custom", @"A comment");
```

Using the Core Foundation framework, you could get the same string using the `CFCopyLocalizedStringFromTable` macro, as shown in this example:

```
CFStringRef theString;
theString = CFCopyLocalizedStringFromTable(CFSTR("Yes"), CFSTR("Custom"), "A
comment");
```

In both examples, the code specifies the key to retrieve, which is the string "Yes". They also specify the strings file (or table) in which to look for the key, which in this case is the `Custom.strings` file. During string retrieval, the comment string is ignored.

Advanced Strings File Tips

The following sections provide some additional tips for working with strings files and string resources.

Searching for Custom Functions With genstrings

The `genstrings` tool searches for the Core Foundation and Foundation string macros by default. It uses the information in these macros to create the string entries in your project's strings files. You can also direct `genstrings` to look for custom string-loading functions in your code and use those functions in addition to the standard macros. You might use custom functions to wrap the built-in string-loading routines and perform some extra processing or you might replace the default string handling behavior with your own custom model.

If you want to use `genstrings` with your own custom functions, your functions must use the naming and formatting conventions used by the Foundation macros. The parameters for your functions must match the parameters for the corresponding macros exactly. When you invoke genstrings, you specify the `-s` option followed by the name of the function that corresponds to the `NSLocalizedString` macro. Your other function names should then build from this base name. For example, if you specified the function name `MyStringFunction`, your other function names should be `MyStringFunctionFromTable`, `MyStringFunctionFromTableInBundle`, and `MyStringFunctionWithDefaultValue`. The `genstrings` tool looks for these functions and uses them to build the corresponding strings files.

Formatting String Resources

For some strings, you may not want to (or be able to) encode the entire string in a string resource because portions of the string might change at runtime. For example, if a string contains the name of a user document, you need to be able to insert that document name into the string dynamically. When creating your string resources, you can use any of the formatting characters you would normally use for handling string replacement in the Foundation and Core Foundation frameworks. Listing 4 shows several string resources that use basic formatting characters:

Listing 4 Strings with formatting characters

```
"Windows must have at least %d columns and %d rows." =
"Les fenêtres doivent être composés au minimum de %d colonnes et %d lignes.";
"File %@ not found." = "Le fichier %@ n'existe pas.";
```

To replace formatting characters with actual values, you use the `stringWithFormat:` method of `NSString` or the `CFStringCreateWithFormat` function, using the string resource as the format string. Foundation and Core Foundation support most of the standard formatting characters used in `printf` statements. In addition, you can use the `%@` specifier shown in the preceding example to insert the descriptive text associated with arbitrary Objective-C objects. See Formatting String Objects in *String Programming Guide for Cocoa* for the complete list of specifiers.

One problem that often occurs during translation is that the translator may need to reorder parameters inside translated strings to account for differences in the source and target languages. If a string contains multiple arguments, the translator can insert special tags of the form *n*`$` (where *n* specifies the position of the original

argument) in between the formatting characters. These tags let the translator reorder the arguments that appear in the original string. The following example shows a string whose two arguments are reversed in the translated string:

```
/* Message in alert dialog when something fails */
"%@ Error! %@ failed!" = "%2$@ blah blah, %1$@ blah!";
```

Using Special Characters in String Resources

Just as in C, some characters must be prefixed with a backslash before you can include them in the string. These characters include double quotation marks, the backslash character itself, and special control characters such as linefeed (`\n`) and carriage returns (`\r`).

```
"File \"%@\" cannot be opened" = " ... ";
"Type \"OK\" when done" = " ... ";
```

You can include arbitrary Unicode characters in a value string by specifying `\\U` followed immediately by up to four hexadecimal digits. The four digits denote the entry for the desired Unicode character; for example, the space character is represented by hexadecimal 20 and thus would be `\\U0020` when specified as a Unicode character. This option is useful if a string must include Unicode characters that for some reason cannot be typed. If you use this option, you must also pass the `-u` option to `genstrings` in order for the hexadecimal digits to be interpreted correctly in the resulting strings file. The `genstrings` tool assumes your strings are low-ASCII by default and only interprets backslash sequences if the `-u` option is specified.

Note: The `genstrings` tool always generates strings files using the UTF-16 encoding. If you include Unicode characters in your strings and do not use `genstrings` to create your strings files, be sure to save your strings files in the UTF-16 encoding.

Debugging Strings Files

If you run into problems during testing and find that the functions and macros for retrieving strings are always returning the same key (as opposed to the translated value), run the `/usr/bin/plutil` tool on your strings file. A strings file is essentially a property-list file formatted in a special way. Running `plutil` with the `-lint` option can uncover hidden characters or other errors that are preventing strings from being retrieved correctly.

Internationalizing Other Resources

User-interface objects such as cells and text fields localize their content automatically when they parse or format data. Thus if you ask a cell for its numerical value, and the user's locale is French, "," will be used as the decimal separator. Similarly, if you display a floating point number in cells, table views, and so on, they are localized automatically.

Resources and Core Foundation

If you are using Core Foundation and need to format or scan date or number information, use the `CFDateFormatterRef` and `CFNumberFormatterRef` types. These types use the current locale information to ensure that dates and numbers are formatted correctly. For more information, see *CFDateFormatter Reference* and *CFNumberFormatter Reference*.

Resources and Cocoa

If you are formatting or scanning dates or numbers yourself using low-level objects such as `NSString`, `NSDate`, or `NSScanner`, you should use the current locale information if the resulting text will be seen by the user. In Mac OS X v10.4, the `NSLocale` class was added to Cocoa to make it easier to get locale information. In addition to that class, several cocoa classes offer methods that provide an explicit locale argument:

```
NSString:
    - (id)initWithFormat:(NSString *)format locale:(NSDictionary *)dict, ...;
    + (id)stringWithFormat:(NSString *)format, ...;
    + (id)localizedStringWithFormat:(NSString *)format, ...;

NSScanner:
    - (void)setLocale:(NSDictionary *)dict;
    + (id)scannerWithString:(NSString *)string;
    + (id)localizedScannerWithString:(NSString *)string;

NSObject:
    - (NSString *)description;
    - (NSString *)descriptionWithLocale:(NSDictionary *)locale;

NSDate:
    - (id)initWithString:(NSString *)description calendarFormat:(NSString *)format
    locale:(NSDictionary *)dict;
```

A locale is represented as a dictionary, using key/value pairs to store information about how the localization should be performed. Some of the possible keys in this dictionary are listed in `Foundation/NSUserDefaults.h`. Typically you pass `nil` or the dictionary built from the standard user defaults. To create the dictionary from the user preferences, you would use the following code:

```
[[NSUserDefaults standardUserDefaults] dictionaryRepresentation]
```

This code returns a dictionary with a flattened view of the user's defaults and languages in order of user preference. The user's defaults take precedence, so any language-specific information might be overridden by entries in defaults (which are typically set from user's preferences).

The Application Kit makes localization easier by providing cover methods that ask for a “localized” version of an object, such as:

```
- (id)initWithFormat:(NSString *)format locale:(NSDictionary *)dict, ...;
+ (id)localizedStringWithFormat:(NSString *)format, ...;
```

The second method calls the first one with a locale argument of

```
[[NSUserDefaults standardUserDefaults] dictionaryRepresentation]
```

In versions of these methods without a locale argument, the processing is done in a non-localized manner.

Localizing Pathnames

In order to provide a better user experience, Mac OS X supports the ability to display localized names for system and application directories. Localized pathnames make it easier for international users to find information, but do so in a way that does not impede your application. The implementation of this feature merely replaces the name of some directories with a localized version when they are displayed to the user. The actual path information on the hard disk does not change.

There are two ways to support localized pathnames. The first is to display them in your application. The second is to localize the names of any directories associated with your application.

The following sections explain the process for getting localized pathnames and for localizing your application's path information. For more information on localizing your application menus and content, see "Preparing Your Nib Files for Localization" (page 33).

Important: Mac OS X does not display localized pathnames for items in the Darwin and Classic environments.

Getting Localized Path Names

You need to be aware of localized path names in your application and display them appropriately. Localized path names are for display only and should never be used to access the file system. For the most part, you should continue to use the actual pathname when working with files and directories in your code, including when you need to write to caches or user preferences. The only time you should use a localized path name is when you want to display that path to the user through your application's user interface.

Mac OS X provides several functions for obtaining the localized name of a path. You should always use one of these functions to convert a path to its localized name immediately prior to display. All applications can use the Launch Services methods `LSCopyDisplayNameForRef` and `LSCopyDisplayNameForURL` to retrieve localized display names. Cocoa applications can also use the `NSFileManager` methods `displayNameAtPath:` and `componentsToDisplayForPath:` to retrieve this information.

Note: These functions do more than just localize path names. They also hide file extensions when called for by the current file and Finder settings. Thus, they provide an abstraction between the actual file system and the file system as presented by the Finder. By using display names, your application helps to enforce this abstraction.

Localizing Your Application Name

If you have a bundled application, you can specify a localized display name for your application. The Finder displays localized bundle names based on the current language settings. Other applications can request your application's localized name as well and display it as appropriate.

Note: Mac OS X does not support localized names for non-bundled applications.

You specify localized names for your application using the existing bundle localization mechanism. The Resources folder of your application bundle contains one or more language-specific resource directories. (See "Anatomy of a Modern Bundle" in *Bundle Programming Guide* for information about bundle resource directories.) In each of these language-specific directories, you can include an `InfoPlist.strings` file with a list of localized property-list keys. One of the keys you can include in this file is the `CFBundleDisplayName` key, whose value you can set to the localized name of your bundle.

At all times, Mac OS X prefers user-customized display names over the default and localized names you specify in your bundle. If the on-disk application name is different than the non-localized version of your bundle display name—that is, the name associated with the `CFBundleDisplayName` key in your `Info.plist` file—the system assumes the user made the change and returns the customized name. If at some later time, the user changes the application name back to the original name, the system reverts to using the localized values from your application bundle.

Important: If you want your localized display names to appear, you must include the `LSHasLocalizedDisplayName` key in your application's `Info.plist` file, set the type of its value to Boolean, and set the value to true. The functions that access localized display name information check for the existence of this key before retrieving the information.

Localizing Directory Names

If your application package installs any custom support directories, you can provide localized names for those directories; however, doing so is not required. If you want to localize your application's support directories, you should do so only for directories whose names are known in advance by your application.

Note: Localized directory names appear only if the "Always show file extensions" option is disabled in the Finder preferences. Localized names do not appear until the next time the user logs in.

To localize a directory name, add the extension `.localized` to the directory name. Inside your directory, use the Terminal application to create a subdirectory called `.localized`. Inside this subdirectory, put one or more strings files corresponding to the localizations you support. Each strings file is a Unicode text file. The name of the file is the appropriate two-letter language code followed by the `.strings` extension. For example, a localized Release Notes directory with English, Japanese, and German localizations would have the following directory structure:

```
Release Notes.localized/
    .localized/
        en.strings
        de.strings
        ja.strings
```

Inside each of strings files, include a single string entry to map the non-localized directory name to the localized name. For example, to map the name "Release Notes" to a localized directory name, each strings file would have an entry similar to the following:

```
"Release Notes" = "Localized name";
```

For information on creating a strings file, see "Creating Strings Resource Files" (page 38).

Note: System-defined directories, such as `/System`, `/Library`, and the default directories in each user's home directory, use a different localization scheme than the one described here. For these directories, the presence of an empty file with the name `.localized` causes the system to display the directory with a localized name. Do not delete the `.localized` file from any these directories.

Notes For Localizers

This task provides some assistance for localizers of strings files and nib files.

Localizing Strings Files

Strings files generated by `genstrings` have content that looks like this:

```
/* Comment */
"key" = "key";
```

That is, the value string is the same as the key string.

Translators should type the localized version of the key string in place of the second string. Use the comment, if necessary, to understand the context in which the string is displayed to the user:

```
/* Comment */
"key" = "French version of the key";
```

If a string contains multiple variable arguments, you can change the order of the arguments by using the "$" modifier plus the argument number

```
/* Message in alert panel when something fails */
"Oh %@! %@ failed!" = "%2$@ blah blah, %1$@ oh!";
```

Just as in C, some characters must be prefixed with a backslash to be included in the string properly. These characters include double-quote, backslash, and carriage return. You can also specify carriage returns with "`\n`":

```
"File \"%@\" cannot be opened" = " ... ";
"Type \"OK\" when done" = " ... ";
```

Strings can include arbitrary Unicode characters with "`\U`" followed by up to four hexadecimal digits denoting the Unicode character; for instance, space, which is hexadecimal 20, is represented as "`\U0020`". This option is useful if strings must include Unicode characters which cannot be typed for some reason.

Strings files are best saved in Unicode format. This allows them to be encoding-independent, and simplifies the encoding to use when an application loads strings files. The TextEdit application can save in Unicode format. The encoding can be selected either from the Save panel, or as a general preference in TextEdit's Preferences panel.

For more information about strings files, see "Strings Files" (page 37).

Localizing Nib Files

You use Interface Builder to create nib files, and you should use Interface Builder to localize nib files. Typically you open all of the nib files in a *language*`.lproj` directory, localize all the strings, change the sizes of UI elements if necessary, and save the nib files. There are a few things to watch out for:

- Objects in a nib file typically have connections between them that should not be broken. You should lock all connections (an option in the Preferences panel of Interface Builder) before editing the nibs.
- Panels and windows usually have minimum or maximum sizes that are specified through the inspector panel. If you must make a panel or window wider for a given language, it's likely that the minimum size also needs to be modified.
- Some UI objects support “tool tips” – little blurbs that come up when the user hovers on the UI element for a short while. You enter these strings in the inspector, where they should also be localized.

File Encodings and Fonts

Unicode is generally considered the native encoding for Mac OS X and should be used in nearly all situations. Previous versions of Mac OS supported file encodings such as MacRoman but most modern Mac OS X libraries support Unicode inherently. If you use Cocoa or Core Foundation routines, then you will probably never need to worry about other file encodings. If your software supports legacy file formats, however, you might need to consider file encoding issues when importing legacy file formats. The following sections describe some of the issues related to Unicode support and legacy file encodings.

File Systems and Unicode Support

Different file systems in Mac OS X have different levels of Unicode support:

- Mac OS Extended (HFS+) uses canonically decomposed Unicode 3.2 in UTF-16 format, which consists of a sequence of 16-bit codes. (Characters in the ranges U2000-U2FFF, UF900-UFA6A, and U2F800-U2FA1D are not decomposed.)
- The UFS file system allows any character from Unicode 2.1 or later, but uses the UTF-8 format, which consists mostly of 8-bit ASCII codes but which may also include multibyte codes. (Characters in the ranges U2000-U2FFF, UF900-UFA6A, and U2F800-U2FA1D are not decomposed.)
- Mac OS Standard (HFS) does not support Unicode and instead uses legacy Mac encodings, such as MacRoman.

Locking the canonical decomposition to a particular version of Unicode does not exclude usage of characters defined in a newer version of Unicode. Because the Unicode consortium has guaranteed not to add any more precomposed characters, applications can expect to store characters defined in future versions of Unicode without compatibility issues.

Note: Because of implementation differences, erroneous Unicode in filenames on HFS+ volumes may display correctly when entered on Mac OS 9 but appear garbled on Mac OS X. Similarly, erroneous Unicode entered on Mac OS X may appear garbled in Mac OS 9.

All BSD system functions expect their string parameters to be in UTF-8 encoding and nothing else. Code that calls BSD system routines should ensure that the contents of all `const *char` parameters are in canonical UTF-8 encoding. In a canonical UTF-8 string, all decomposable characters are decomposed; for example, é (0x00E9) is represented as e (0x0065) + ´ (0x0301). To put things into a canonical UTF-8 encoding, use the "file-system representation" interfaces defined in Cocoa and Carbon (including Core Foundation).

Getting Canonical Strings

Both Cocoa and Core Foundation provide routines for accessing canonical and non-canonical Unicode strings. Cocoa string manipulations are all handled through the `NSString` class and its subclasses. In Core Foundation, you can use the `CFStringGetCString` and `CFStringGetCStringPtr` functions to obtain a C string with the desired encoding.

Carbon and QuickDraw Issues

If you have existing QuickDraw code and want to draw text, you should be aware that the QuickDraw Text routines do not directly support Unicode. The Carbon File Manager has some file-system calls that return Mac encodings and others that return Unicode. If you pass this Unicode text directly to a QuickDraw routine, you may run into problems. Similarly, if you retrieve text in a Mac encoding and want to use it with Cocoa or with Carbon's Apple Type Services for Unicode Imaging (ATSUI) API, you must convert the text to Unicode first.

Generally, the encoding that is used depends upon the API you use and not on the font. Fonts are not necessarily limited to particular encodings. TrueType fonts, for example, declare the set of glyphs they implement and provide encoding tables that map those glyphs to character values in particular encodings. PostScript fonts have similar encoding tables. Various parts of the operating system know how to map characters from one encoding to another. Cocoa and ATSUI use Unicode as the "destination" mapping for a font. QuickDraw Text in Carbon uses the Mac encodings, selected according to the script that the 'FOND' resource of the font corresponds to.

The fonts that are installed with Mac OS X have large character sets supporting a wide range of encodings and scripts. For example, Lucida, the system font, supports extended Latin, Greek, Cyrillic, Arabic, Hebrew, and Thai. But if you draw text through QuickDraw Text, you have access only to the MacRoman repertoire. To access the rest, you must use Cocoa or ATSUI. Similarly, the Hiragino fonts also have a large repertoire of characters beyond that supported by MacJapanese, and these are accessible only through Cocoa or ATSUI. Both Cocoa and ATSUI also substitute glyphs from other fonts when the requested one isn't available; however, their algorithms for font substitution are different.

For information on file encodings in the context of multiscript support, see "Guidelines for Adding MultiScript Support" (page 26).

Cocoa Issues

Cocoa employs Unicode for character encoding, making any Cocoa application capable of displaying most human languages. Although Cocoa supports vertical and bidirectional text, the `NSTypesetter` class only supports layout for horizontal text. If you want to lay out vertical text, you need to define your own custom typesetter class.

Document Revision History

This table describes the changes to *Internationalization Programming Topics*.

Date	Notes
2009-01-06	Fixed the ibtool examples to use the updated syntax.
2008-09-09	Updated nibtool references to ibtool, which is the new name for the tool. Fixed some additional bugs.
2005-09-08	Merged the Cocoa Internationalization document into this document.
	Merged information from several articles aboug creating and managing strings files into one article.
	Added references to CFLocale and NSLocale.
	Updated the guidelines related to the use of language and locale IDs.
	Added information about how to use `nibtool` and `plutil`.
	Changed the title from "Internationalizing Your Software".
2005-03-03	Updated information on genstrings and usage of UTF-8 and UTF-16 in strings files.
2004-04-15	Added an article containing general guidelines. Moved the MultiScript guidelines to this article.
	Added an article on how to get the current localization information.
	Updated the language designation article to reflect the recommended use of ISO standards for designating language and region information.
	Fixed minor bugs.
2003-08-07	First revision of this programming topic. Some information in this programming topic originally appeared in *System Overview*.

Index

M

N

O

P

Q

S

T

U

W

X