Localization

Android will run on many devices in many regions. To reach the most users, your application should handle text, audio files, numbers, currency, and graphics in ways appropriate to the locales where your application will be used.

This document describes best practices for localizing Android applications. The principles apply whether you are developing your application using ADT with Eclipse, Ant-based tools, or any other IDE.

You should already have a working knowledge of Java and be familiar with Android resource loading, the declaration of user interface elements in XML, development considerations such as Activity lifecycle, and general principles of internationalization and localization.

It is good practice to use the Android resource framework to separate the localized aspects of your application as much as possible from the core Java functionality:

- You can put most or all of the *contents* of your application's user interface into resource files, as described in this document and in Providing Resources.
- The *behavior* of the user interface, on the other hand, is driven by your Java code. For example, if users input data that needs to be formatted or sorted differently depending on locale, then you would use Java to handle the data programmatically. This document does not cover how to localize your Java code.

The Hello, L10N tutorial takes you through the steps of creating a simple localized application that uses locale-specific resources in the way described in this document.

Localization quickview

- Android lets you create different resource sets for different locales.
- When your application runs, Android will load the resource set that match the device's locale.
- If locale-specific resources are not available, Android falls back to defaults.
- The emulator has features for testing localized apps.

In this document

See also

Hello, L10N Tutorial
Providing Resources
Declaring Layout
Activity Lifecycle

Overview: Resource-Switching in Android

Resources are text strings, layouts, sounds, graphics, and any other static data that your Android application needs. An application can include multiple sets of resources, each customized for a different device configuration. When a user runs the application, Android automatically selects and loads the resources that best match the device.

(This document focuses on localization and locale. For a complete description of resource-switching and all the types of configurations that you can specify — screen orientation, touchscreen type, and so on — see Providing Alternative Resources.)

When you write your application:	→	When a user runs your application:
You create a set of default resources, plus alternatives to be used in different locales.		The Android system selects which resources to load, based on the device's locale.

When you write your application, you create default and alternative resources for your application to use. To create resources, you place files within specially named subdirectories of the project's `res/` directory.

Why Default Resources Are Important

Whenever the application runs in a locale for which you have not provided locale-specific text, Android will load the default strings from `res/values/strings.xml`. If this default file is absent, or if it is missing a string that your application needs, then your application will not run and will show an error. The example below illustrates what can happen when the default text file is incomplete.

Example:

An application's Java code refers to just two strings, `text_a` and `text_b`. This application includes a localized resource file (`res/values-en/strings.xml`) that defines `text_a` and `text_b` in English. This application also includes a default resource file (`res/values/strings.xml`) that includes a definition for `text_a`, but not for `text_b`:

- This application might compile without a problem. An IDE such as Eclipse will not highlight any errors if a resource is missing.
- When this application is launched on a device with locale set to English, the application might run without a problem, because `res/values-en/strings.xml` contains both of the needed text strings.
- However, **the user will see an error message and a Force Close button** when this application is launched on a device set to a language other than English. The application will not load.

To prevent this situation, make sure that a `res/values/strings.xml` file exists and that it defines every needed string. The situation applies to all types of resources, not just strings: You need to create a set of default resource files containing all the resources that your application calls upon — layouts, drawables, animations, etc. For information about testing, see Testing for Default Resources.

Using Resources for Localization

How to Create Default Resources

Put the application's default text in a file with the following location and name:

`res/values/strings.xml` (required directory)

The text strings in `res/values/strings.xml` should use the default language, which is the language that you expect most of your application's users to speak.

The default resource set must also include any default drawables and layouts, and can include other types of resources such as animations.
`res/drawable/`(required directory holding at least one graphic file, for the application's icon in the Market)
`res/layout/` (required directory holding an XML file that defines the default layout)
`res/anim/` (required if you have any `res/anim-<qualifiers>` folders)
`res/xml/` (required if you have any `res/xml-<qualifiers>` folders)
`res/raw/` (required if you have any `res/raw-<qualifiers>` folders)

> **Tip:** In your code, examine each reference to an Android resource. Make sure that a default resource is defined for each one. Also make sure that the default string file is complete: A *localized* string file can contain a subset of the strings, but the *default* string file must contain them all.

How to Create Alternative Resources

A large part of localizing an application is providing alternative text for different languages. In some cases you will also provide alternative graphics, sounds, layouts, and other locale-specific resources.

An application can specify many `res/<qualifiers>/` directories, each with different qualifiers. To create an alternative resource for a different locale, you use a qualifier that specifies a language or a language-region combination. (The name of a resource directory must conform to the naming scheme described in Providing Alternative Resources, or else it will not compile.)

Example:

Suppose that your application's default language is English. Suppose also that you want to localize all the text in your application to French, and most of the text in your application (everything except the application's title) to Japanese. In this case, you could create three alternative `strings.xml` files, each stored in a locale-specific resource directory:

1. `res/values/strings.xml`
   Contains English text for all the strings that the application uses, including text for a string named `title`.
2. `res/values-fr/strings.xml`
   Contain French text for all the strings, including `title`.
3. `res/values-ja/strings.xml`
   Contain Japanese text for all the strings *except* `title`.

If your Java code refers to `R.string.title`, here is what will happen at runtime:

- If the device is set to any language other than French, Android will load `title` from the `res/values/strings.xml` file.
- If the device is set to French, Android will load `title` from the `res/values-fr/strings.xml` file.

Notice that if the device is set to Japanese, Android will look for `title` in the `res/values-ja/strings.xml` file. But because no such string is included in that file, Android will fall back to the default, and will load `title` in English from the `res/values/strings.xml` file.

Which Resources Take Precedence?

If multiple resource files match a device's configuration, Android follows a set of rules in deciding which file to use. Among the qualifiers that can be specified in a resource directory name, **locale almost always takes precedence**.

Example:

Assume that an application includes a default set of graphics and two other sets of graphics, each optimized for a different device setup:

- `res/drawable/`
  Contains default graphics.
- `res/drawable-small-land-stylus/`
  Contains graphics optimized for use with a device that expects input from a stylus and has a QVGA low-density screen in landscape orientation.
- `res/drawable-ja/`
  Contains graphics optimized for use with Japanese.

If the application runs on a device that is configured to use Japanese, Android will load graphics from `res/drawable-ja/`, even if the device happens to be one that expects input from a stylus and has a QVGA low-density screen in landscape orientation.

> **Exception:** The only qualifiers that take precedence over locale in the selection process are MCC and MNC (mobile country code and mobile network code).

Example:

Assume that you have the following situation:

- The application code calls for `R.string.text_a`
- Two relevant resource files are available:
  - `res/values-mcc404/strings.xml`, which includes `text_a` in the application's default language, in this case English.

    - `res/values-hi/strings.xml`, which includes `text_a` in Hindi.
- The application is running on a device that has the following configuration:
    - The SIM card is connected to a mobile network in India (MCC 404).
    - The language is set to Hindi (`hi`).

Android will load `text_a` from `res/values-mcc404/strings.xml` (in English), even if the device is configured for Hindi. That is because in the resource-selection process, Android will prefer an MCC match over a language match.

The selection process is not always as straightforward as these examples suggest. Please read How Android Finds the Best-matching Resource for a more nuanced description of the process. All the qualifiers are described and listed in order of precedence in Table 2 of Providing Alternative Resources.

Referring to Resources in Java

In your application's Java code, you refer to resources using the syntax `R.`*`resource_type`*`.`*`resource_name`* or `android.R.`*`resource_type`*`.`*`resource_name`*. For more about this, see Accessing Resources.

Localization Strategies

Design your application to work in any locale

You cannot assume anything about the device on which a user will run your application. The device might have hardware that you were not anticipating, or it might be set to a locale that you did not plan for or that you cannot test. Design your application so that it will function normally or fail gracefully no matter what device it runs on.

> **Important:** Make sure that your application includes a full set of default resources.

Make sure to include `res/drawable/` and a `res/values/` folders (without any additional modifiers in the folder names) that contain all the images and text that your application will need.

If an application is missing even one default resource, it will not run on a device that is set to an unsupported locale. For example, the `res/values/strings.xml` default file might lack one string that the application needs: When the application runs in an unsupported locale and attempts to load `res/values/strings.xml`, the user will see an error message and a Force Close button. An IDE such as Eclipse will not highlight this kind of error, and you will not see the problem when you test the application on a device or emulator that is set to a supported locale.

For more information, see Testing for Default Resources.

Design a flexible layout

If you need to rearrange your layout to fit a certain language (for example German with its long words), you can create an alternative layout for that language (for example `res/layout-de/main.xml`). However, doing this can make your application harder to maintain. It is better to create a single layout that is more flexible.

Another typical situation is a language that requires something different in its layout. For example, you might have a contact form that should include two name fields when the application runs in Japanese, but three name fields when the application runs in some other language. You could handle this in either of two ways:

- Create one layout with a field that you can programmatically enable or disable, based on the language, or
- Have the main layout include another layout that includes the changeable field. The second layout can have different configurations for different languages.

Avoid creating more resource files and text strings than you need

You probably do not need to create a locale-specific alternative for every resource in your application. For example, the layout defined in the `res/layout/main.xml` file might work in any locale, in which case there would be no need to create any alternative layout files.

Also, you might not need to create alternative text for every string. For example, assume the following:

- Your application's default language is American English. Every string that the application uses is defined, using

American English spellings, in `res/values/strings.xml`.

- For a few important phrases, you want to provide British English spelling. You want these alternative strings to be used when your application runs on a device in the United Kingdom.

To do this, you could create a small file called `res/values-en-rGB/strings.xml` that includes only the strings that should be different when the application runs in the U.K. For all the rest of the strings, the application will fall back to the defaults and use what is defined in `res/values/strings.xml`.

Use the Android Context object for manual locale lookup

You can look up the locale using the `Context` object that Android makes available:

```
String locale = context.getResources().getConfiguration().locale.getDisplayName();
```

Testing Localized Applications

Testing on a Device

Keep in mind that the device you are testing may be significantly different from the devices available to consumers in other geographies. The locales available on your device may differ from those available on other devices. Also, the resolution and density of the device screen may differ, which could affect the display of strings and drawables in your UI.

To change the locale on a device, use the Settings application (Home > Menu > Settings > Locale & text > Select locale).

Testing on an Emulator

For details about using the emulator, see See Android Emulator.

Creating and using a custom locale

A "custom" locale is a language/region combination that the Android system image does not explicitly support. (For a list of supported locales in Android platforms see the Version Notes in the SDK tab). You can test how your application will run in a custom locale by creating a custom locale in the emulator. There are two ways to do this:

- Use the Custom Locale application, which is accessible from the Application tab. (After you create a custom locale, switch to it by pressing and holding the locale name.)
- Change to a custom locale from the adb shell, as described below.

When you set the emulator to a locale that is not available in the Android system image, the system itself will display in its default language. Your application, however, should localize properly.

Changing the emulator locale from the adb shell

To change the locale in the emulator by using the adb shell.

1. Pick the locale you want to test and determine its language and region codes, for example `fr` for French and `CA` for Canada.
2. Launch an emulator.
3. From a command-line shell on the host computer, run the following command:
   `adb shell`
   or if you have a device attached, specify that you want the emulator by adding the `-e` option:
   `adb -e shell`
4. At the adb shell prompt (`#`), run this command:
   `setprop persist.sys.language [`*`language code`*`];setprop persist.sys.country [`*`country code`*`];stop;sleep 5;start`
   Replace bracketed sections with the appropriate codes from Step 1.

For instance, to test in Canadian French:

```
setprop persist.sys.language fr;setprop persist.sys.country CA;stop;sleep 5;start
```

This will cause the emulator to restart. (It will look like a full reboot, but it is not.) Once the Home screen appears again, re-launch your application (for example, click the Run icon in Eclipse), and the application will launch with the new locale.

Testing for Default Resources

Here's how to test whether an application includes every string resource that it needs:

1. Set the emulator or device to a language that your application does not support. For example, if the application has French strings in `res/values-fr/` but does not have any Spanish strings in `res/values-es/`, then set the emulator's locale to Spanish. (You can use the Custom Locale application to set the emulator to an unsupported locale.)
2. Run the application.
3. If the application shows an error message and a Force Close button, it might be looking for a string that is not available. Make sure that your `res/values/strings.xml` file includes a definition for every string that the application uses.

If the test is successful, repeat it for other types of configurations. For example, if the application has a layout file called `res/layout-land/main.xml` but does not contain a file called `res/layout-port/main.xml`, then set the emulator or device to portrait orientation and see if the application will run.

Publishing Localized Applications

The Android Market is the main application distribution system for Android devices. To publish a localized application, you need to sign your application, version it, and go through all the other steps described in Preparing to Publish.

If you split your application in several .apk files, each targeted to a different locale, follow these guidelines:

- Sign each .apk file with the same certificate. For more about this, see Signing Strategies.
- Give each .apk file a different application name. Currently it is impossible to put two applications into the Android Market that have exactly the same name.
- Include a complete set of default resources in each .apk file.

Localization Checklists

These checklists summarize the process of localizing an Android application. Not everything on these lists will apply to every application.

Planning and Design Checklist

→	Choose a localization strategy. Which countries and which languages will your application support? What is your application's default country and language? How will your application behave when it does not have specific resources available for a given locale?
→	Identify everything in your application that will need to be localized: • Consider specific details of your application — text, images, sounds, music, numbers, money, dates and times. You might not need to localize everything. For example, you don't need to localize text that the user never sees, or images that are culturally neutral, or icons that convey the same meaning in every locale. • Consider broad themes. For example, if you hope to sell your application in two very culturally different

	markets, you might want to design your UI and present your application in an entirely different way for each locale.
→	Design your Java code to externalize resources wherever possible: • Use `R.string` and `strings.xml` files instead of hard-coded strings or string constants. • Use `R.drawable` and `R.layout` instead of hard-coded drawables or layouts.

Content Checklist

→	Create a full set of default resources in `res/values/` and other `res/` folders, as described in Creating Default Resources.
→	Obtain reliable translations of the static text, including menu text, button names, error messages, and help text. Place the translated strings in `res/values-<qualifiers>/strings.xml` files.
→	Make sure that your application correctly formats dynamic text (for example numbers and dates) for each supported locale. Make sure that your application handles word breaks, punctuation, and alphabetical sorting correctly for each supported language.
→	If necessary, create locale-specific versions of your graphics and layout, and place them in `res/drawable-<qualifiers>/` and `res/layout-<qualifiers>/` folders.
→	Create any other localized content that your application requires; for example, create recordings of sound files for each language, as needed.

Testing and Publishing Checklist

→	Test your application for each supported locale. If possible, have a person who is native to each locale test your application and give you feedback.
→	Test the default resources by loading a locale that is not available on the device or emulator. For instructions, see Testing for Default Resources.
→	Test the localized strings in both landscape and portrait display modes.
→	Sign your application and create your final build or builds.
→	Upload your .apk file or files to Market, selecting the appropriate languages as you upload. (For more details, see Publishing Your Applications.)

← Back to Application Resources ↑ Go to top

Except as noted, this content is licensed under Apache 2.0. For details and restrictions, see the Content License.
Android 3.1 r1 - 17 Jun 2011 10:58
Site Terms of Service - Privacy Policy - Brand Guidelines