Creating an Input Method

In This Document

See also

- Onscreen Input Methods

Sample

- SoftKeyboard

An input method editor (IME) is a user control that enables users to enter text. Android provides an extensible input-method framework that allows applications to provide users alternative input methods, such as on-screen keyboards or even speech input. After installing the desired IMEs, a user can select which one to use from the system settings, and use it across the entire system; only one IME may be enabled at a time.

To add an IME to the Android system, you create an Android application containing a class that extends `InputMethodService`. In addition, you usually create a "settings" activity that passes options to the IME service. You can also define a settings UI that's displayed as part of the system settings.

This guide covers the following:

- The IME lifecycle
- Declaring IME components in the application manifest
- The IME API
- Designing an IME UI
- Sending text from an IME to an application
- Working with IME subtypes

If you haven't worked with IMEs before, you should read the introductory article Onscreen Input Methods first. Also, the SoftKeyboard sample app included in the SDK contains sample code that you can modify to start building your own IME.

The IME Lifecycle

The following diagram describes the life cycle of an IME:

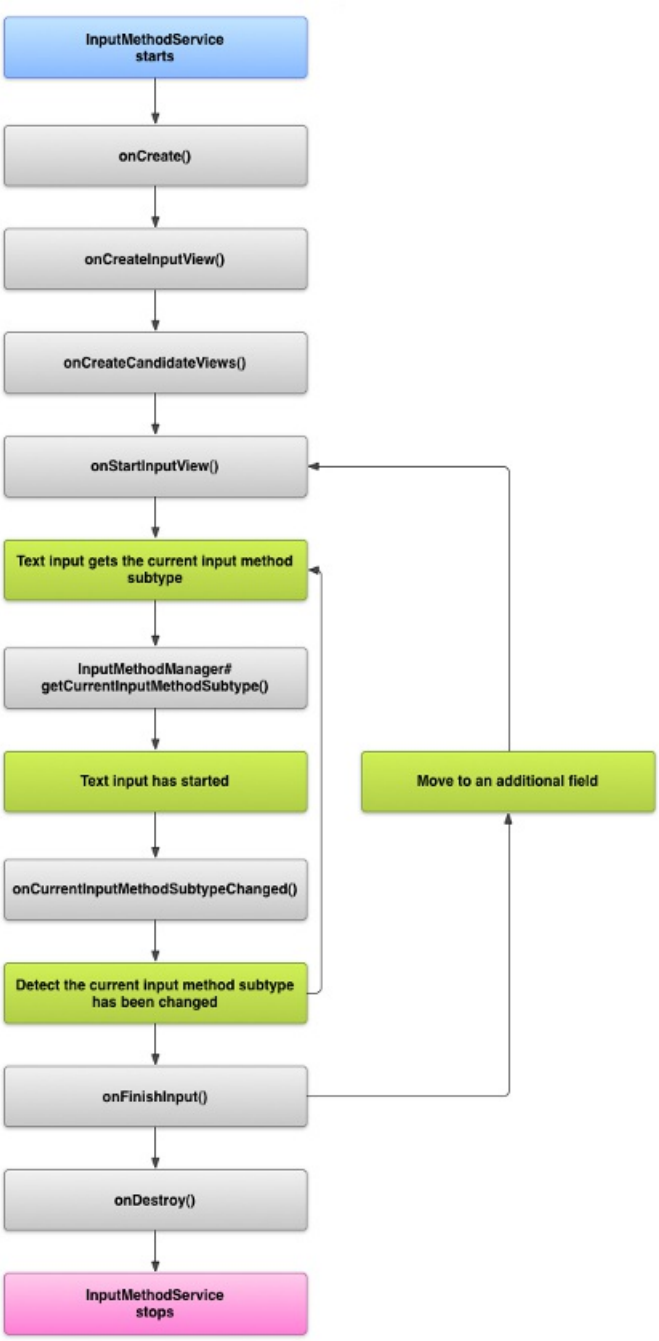

Figure 1. The life cycle of an IME.

The following sections describe how to implement the UI and code associated with an IME that follows this lifecycle.

Declaring IME Components in the Manifest

In the Android system, an IME is an Android application that contains a special IME service. The application's manifest file must declare the service, request the necessary permissions, provide an intent filter that matches the action `action.view.InputMethod`, and provide metadata that defines characteristics of the IME. In addition, to provide a settings interface that allows the user to modify the behavior of the IME, you can define a "settings" activity that can be launched from System Settings.

The following snippet declares an IME service. It requests the permission `BIND_INPUT_METHOD` to allow the service to connect the IME to the system, sets up an intent filter that matches the action `android.view.InputMethod`, and defines metadata for the IME:

```
<!-- Declares the input method service -->
    <service android:name="FastInputIME"
        android:label="@string/fast_input_label"
        android:permission="android.permission.BIND_INPUT_METHOD">
        <intent-filter>
            <action android:name="android.view.InputMethod" />
        </intent-filter>
        <meta-data android:name="android.view.im"
android:resource="@xml/method" />
    </service>
```

This next snippet declares the settings activity for the IME. It has an intent filter for `ACTION_MAIN` that indicates this activity is the main entry point for the IME application:

```
    <!-- Optional: an activity for controlling the IME settings -->
    <activity android:name="FastInputIMESettings"
        android:label="@string/fast_input_settings">
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
        </intent-filter>
    </activity>
```

You can also provide access to the IME's settings directly from its UI.

The Input Method API

Classes specific to IMEs are found in the `android.inputmethodservice` and `android.view.inputmethod` packages. The `KeyEvent` class is important for handling keyboard characters.

The central part of an IME is a service component, a class that extends `InputMethodService`. In addition to implementing the normal service lifecycle, this class has callbacks for providing your IME's UI, handling user input, and delivering text to the field that currently has focus. By default, the `InputMethodService` class provides most of the implementation for managing the state and visibility of the IME and communicating with the current input field.

The following classes are also important:

`BaseInputConnection`

> Defines the communication channel from an `InputMethod` back to the application that is receiving its input. You use it to read text around the cursor, commit text to the text box, and send raw key events to the application. Applications should extend this class rather than implementing the base interface `InputConnection`.

`KeyboardView`

> An extension of `View` that renders a keyboard and responds to user input events. The keyboard layout is specified by an instance of `Keyboard`, which you can define in an XML file.

Designing the Input Method UI

There are two main visual elements for an IME: the **input** view and the **candidates** view. You only have to implement the elements that are relevant to the input method you're designing.

Input view

The input view is the UI where the user inputs text in the form of keyclicks, handwriting or gestures. When the IME is displayed for the first time, the system calls the `onCreateInputView()` callback. In your implementation of this method, you create the layout you want to display in the IME window and return the layout to the system. This snippet is an example of implementing the `onCreateInputView()` method:

```
@Override
public View onCreateInputView() {
    MyKeyboardView inputView =
        (MyKeyboardView) getLayoutInflater().inflate(R.layout.input, null);

    inputView.setOnKeyboardActionListener(this);
    inputView.setKeyboard(mLatinKeyboard);

    return mInputView;
}
```

In this example, `MyKeyboardView` is an instance of a custom implementation of `KeyboardView` that renders a `Keyboard`. If you're building a traditional QWERTY keyboard, see the KeyboardView class.

Candidates view

The candidates view is the UI where the IME displays potential word corrections or suggestions for the user to select. In the IME lifecycle, the system calls `onCreateCandidatesView()` when it's ready to display the candidates view. In your implementation of this method, return a layout that shows word suggestions, or return null if you don't want to show anything. A null response is the default behavior, so you don't have to implement this if you don't provide suggestions.

For an example implementation that provides user suggestions, see the SoftKeyboard sample app.

UI design considerations

This section describes some specific UI design considerations for IMEs.

Handling multiple screen sizes

The UI for your IME must be able to scale for different screen sizes, and it also must handle both landscape and portrait orientations. In non-fullscreen IME mode, leave sufficient space for the application to show the text field and any associated context, so that no more than half the screen is occupied by the IME. In fullscreen IME mode this is not an issue.

Handling different input types

Android text fields allow you to set a specific input type, such as free-form text, numbers, URLs, email addresses, and search strings. When you implement a new IME, you need to detect the input type of each field and provide the appropriate interface for it. However, you don't have to set up your IME to check that the user entered valid text for the input type; that's the responsibility of the application that owns the text field.

For example, here are screenshots of the interfaces that the Latin IME provided with the Android platform provides for text and phone number inputs:

Figure 2. Latin IME input types.

When an input field receives focus and your IME starts, the system calls `onStartInputView()`, passing in an `EditorInfo` object that contains details about the input type and other attributes of the text field. In this object, the `inputType` field contains the text field's input type.

The `inputType` field is an `int` that contains bit patterns for various input type settings. To test it for the text field's input type, mask it with the constant `TYPE_MASK_CLASS`, like this:

```
inputType & InputType.TYPE_MASK_CLASS
```

The input type bit pattern can have one of several values, including:

`TYPE_CLASS_NUMBER`

A text field for entering numbers. As illustrated in the previous screen shot, the Latin IME displays a number pad for fields of this type.

`TYPE_CLASS_DATETIME`

A text field for entering a date and time.

`TYPE_CLASS_PHONE`

A text field for entering telephone numbers.

`TYPE_CLASS_TEXT`

A text field for entering all supported characters.

These constants are described in more detail in the reference documentation for `InputType`.

The `inputType` field can contain other bits that indicate a variant of the text field type, such as:

`TYPE_TEXT_VARIATION_PASSWORD`

A variant of `TYPE_CLASS_TEXT` for entering passwords. The input method will display dingbats instead of the actual text.

`TYPE_TEXT_VARIATION_URI`

A variant of `TYPE_CLASS_TEXT` for entering web URLs and other Uniform Resource Identifiers (URIs).

`TYPE_TEXT_FLAG_AUTO_COMPLETE`

A variant of `TYPE_CLASS_TEXT` for entering text that the application "auto-completes" from a dictionary, search, or other facility.

Remember to mask `inputType` with the appropriate constant when you test for these variants. The available mask constants are listed in the reference documentation for `InputType`.

> **Caution:** In your own IME, make sure you handle text correctly when you send it to a password field. Hide the password in your UI both in the input view and in the candidates view. Also remember that you shouldn't store passwords on a device. To learn more, see the Designing for Security guide.

Sending Text to the Application

As the user inputs text with your IME, you can send text to the application by sending individual key events or by editing the text around the cursor in the application's text field. In either case, you use an instance of `InputConnection` to deliver the text. To get this instance, call `InputMethodService.getCurrentInputConnection()`.

Editing the text around the cursor

When you're handling the editing of existing text in a text field, some of the more useful methods in `BaseInputConnection` are:

`getTextBeforeCursor()`

Returns a `CharSequence` containing the number of requested characters before the current cursor position.

`getTextAfterCursor()`

Returns a `CharSequence` containing the number of requested characters following the current cursor position.

`deleteSurroundingText()`

Deletes the specified number of characters before and following the current cursor position.

`commitText()`

Commit a `CharSequence` to the text field and set a new cursor position.

For example, the following snippet shows how to replace the four characters to the left of the cursor with the text "Hello!":

```
InputConnection ic = getCurrentInputConnection();
ic.deleteSurroundingText(4, 0);
ic.commitText("Hello", 1);
ic.commitText("!", 1);
```

Composing text before committing

If your IME does text prediction or requires multiple steps to compose a glyph or word, you can show the progress in the text field until the user commits the word, and then you can replace the partial composition with the completed text. You may give special treatment to the text by adding a "span" to it when you pass it to `setComposingText()`.

The following snippet shows how to show progress in a text field:

```
InputConnection ic = getCurrentInputConnection();
ic.setComposingText("Composi", 1);
ic.setComposingText("Composin", 1);
ic.commitText("Composing ", 1);
```

The following screenshots show how this appears to the user:

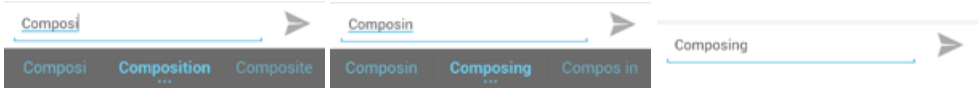

Figure 3. Composing text before committing.

Intercepting hardware key events

Even though the input method window doesn't have explicit focus, it receives hardware key events first and can choose to consume them or forward them along to the application. For example, you may want to consume the directional keys to navigate within your UI for candidate selection during composition. You may also want to trap the back key to dismiss any popups originating from the input method window.

To intercept hardware keys, override `onKeyDown()` and `onKeyUp()`. See the SoftKeyboard sample app for an example.

Remember to call the `super()` method for keys you don't want to handle yourself.

Creating an IME Subtype

Subtypes allow the IME to expose multiple input modes and languages supported by an IME. A subtype can represent:

- A locale such as en_US or fr_FR.
- An input mode such as voice, keyboard, or handwriting.
- Other input styles, forms, or properties specific to the IME, such as 10-key or qwerty keyboard layouts.

Basically, the mode can be any text such as "keyboard", "voice", and so forth. A subtype can also expose a combination of these.

Subtype information is used for an IME switcher dialog that's available from the notification bar and also for IME settings. The information also allows the framework to bring up a specific subtype of an IME directly. When you build an IME, use the subtype facility, because it helps the user identify and switch between different IME languages and modes.

You define subtypes in one of the input method's XML resource files, using the `<subtype>` element. The following snippet defines an IME with two subtypes: a keyboard subtype for the US English locale, and another keyboard subtype for the French language locale for France:

```
<input-method xmlns:android="http://schemas.android.com/apk/res/android"
        android:settingsActivity="com.example.softkeyboard.Settings"
        android:icon="@drawable/ime_icon">
    <subtype android:name="@string/display_name_english_keyboard_ime"
            android:icon="@drawable/subtype_icon_english_keyboard_ime"
            android:imeSubtypeLanguage="en_US"
            android:imeSubtypeMode="keyboard"
            android:imeSubtypeExtraValue="somePrivateOption=true" />
    <subtype android:name="@string/display_name_french_keyboard_ime"
            android:icon="@drawable/subtype_icon_french_keyboard_ime"
            android:imeSubtypeLanguage="fr_FR"
            android:imeSubtypeMode="keyboard"
            android:imeSubtypeExtraValue="foobar=30,someInternalOption=false" />
    <subtype android:name="@string/display_name_german_keyboard_ime" ... />
</input-method>
```

To ensure that your subtypes are labeled correctly in the UI, use %s to get a subtype label that is the same as the subtype's locale label. This is demonstrated in the next two snippets. The first snippet shows part of the input method's XML file:

```
<subtype
    android:label="@string/label_subtype_generic"
    android:imeSubtypeLocale="en_US"
    android:icon="@drawable/icon_en_us"
    android:imeSubtypeMode="keyboard" />
```

The next snippet is part of the IME's `strings.xml` file. The string resource `label_subtype_generic`, which is used by the input method UI definition to set the subtype's label, is defined as:

```
<string name="label_subtype_generic">%s</string>
```

This setting causes the subtype's display name to match the locale setting. For example, in any English locale, the display name is "English (United States)".

Choosing IME subtypes from the notification bar

The Android system manages all subtypes exposed by all IMEs. IME subtypes are treated as modes of the IME they belong to. In the notification bar, a user can select an available subtype for the currently-set IME, as shown in the following screenshot:

Figure 4. Choosing an IME subtype from the notification bar.

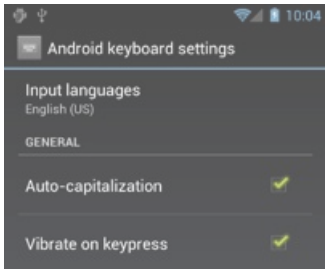

Figure 5. Setting subtype preferences in System Settings.

Choosing IME subtypes from System Settings

A user can control how subtypes are used in the "Language & input" settings panel in the System Settings area. In the SoftKeyboard sample app, the file `InputMethodSettingsFragment.java` contains an implementation that facilitates a subtype enabler in the IME settings. Refer to the SoftKeyboard sample app in the Android SDK for more information about how to support Input Method Subtypes in your IME.

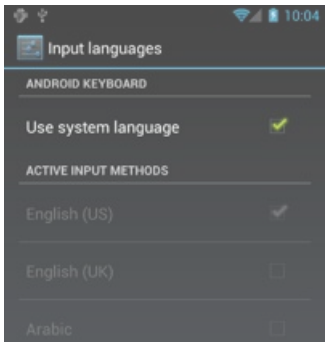

Figure 6. Choosing a language for the IME.

Switching among IME Subtypes

You can allow users to switch easily among multiple IME subtypes by providing a switching key, such as the globe-shaped language icon, as part of the keyboard. Doing so greatly improves the keyboard's usability, and can help avoid user frustration. To enable such switching, perform the following steps:

1. Declare `supportsSwitchingToNextInputMethod = "true"` in the input method's XML resource files. Your declaration should look similar to the following snippet:

   ```
   <input-method xmlns:android="http://schemas.android.com/apk/res/android"
           android:settingsActivity="com.example.softkeyboard.Settings"
           android:icon="@drawable/ime_icon"
           android:supportsSwitchingToNextInputMethod="true">
   ```

2. Call the `shouldOfferSwitchingToNextInputMethod()` method.
3. If the method returns true, display a switching key.
4. When the user taps the switching key, call `switchToNextInputMethod()`, passing false to the second parameter. A value of false tells the system to treat all subtypes equally, regardless of what IME they belong to. Specifying true requires the system to cycle through subtypes in the current IME.

> **Caution:** Prior to Android 5.0 (API level 21), `switchToNextInputMethod()` is not aware of the `supportsSwitchingToNextInputMethod` attribute. If the user switches into an IME without a switching key, he or she may get stuck in that IME, unable to switch out of it easily.

General IME Considerations

Here are some other things to consider as you're implementing your IME:

- Provide a way for users to set options directly from the IME's UI.
- Because multiple IMEs may be installed on the device, provide a way for the user to switch to a different IME directly from the input method UI.
- Bring up the IME's UI quickly. Preload or load on demand any large resources so that users see the IME as soon as they tap on a text field. Cache resources and views for subsequent invocations of the input method.
- Conversely, you should release large memory allocations soon after the input method window is hidden, so that applications can have sufficient memory to run. Consider using a delayed message to release resources if the IME is in a hidden state for a few seconds.
- Make sure that users can enter as many characters as possible for the language or locale associated with the IME. Remember that users may use punctuation in passwords or user names, so your IME has to provide many different characters to allow users to enter a password and get access to the device.