# Basic Customization Guide

J2ME Wireless Toolkit

2.2

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, California 95054
U.S.A.
1-800-555-9SUN or 1-650-960-1300

October 2004

Please
Recycle

# Contents

# Preface

The *J2ME Wireless Toolkit Basic Customization Guide* describes how to create your own device skins, create obfuscator plug-ins and perform other customizations on the J2ME Wireless Toolkit.

## Who Should Use This Book

This guide is intended for developers who need to configure the J2ME Wireless Toolkit to accommodate new device emulator skins. This document assumes that you are familiar with Java programming, Mobile Information Device Profile (MIDP) and the Connected Limited Device Configuration (CLDC) specifications.

## How This Book Is Organized

This guide contains the following chapters and appendixes:

Chapter 1 outlines the possibilities of toolkit customization.

Chapter 2 is a tutorial that shows how to create device property files. The tutorial shows you how to obtain and enter image files, screen properties, button properties, soft button label areas, and icon properties. The tutorial also explains how to set color properties and how to run the emulator for the new device.

Chapter 3 shows how to create a plug-in for an obfuscator.

# Typographic Conventions

| Typeface | Meaning | Examples |
|---|---|---|
| `AaBbCc123` | The names of commands, files, and directories; on-screen computer output | Edit your `.login` file.<br>Use `ls -a` to list all files.<br>`% You have mail.` |
| **`AaBbCc123`** | What you type, when contrasted with on-screen computer output | `% su`<br>`Password:` |
| *AaBbCc123* | Book titles, new words or terms, words to be emphasized | Read Chapter 6 in the *User's Guide*.<br>These are called *class* options.<br>You *must* be superuser to do this. |
| | Command-line variable; replace with a real name or value | To delete a file, type `rm` *filename*. |
| {*AaBbCc.dir*} | Variable file names and directories. | These files are located under the {*toolkit*}`\apps\`{*demo_name*}`\bin\` directory where {*toolkit*} is the installation directory of the J2ME Wireless Toolkit and {*demo_name*} is the name of one of the demo applications. |

# Related Documentation

| Application | Title |
|---|---|
| J2ME Wireless Toolkit | *J2ME Wireless Toolkit User's Guide* |
| J2ME Wireless Toolkit | *J2ME Wireless Toolkit Toolkit Release Notes* |

# Accessing Documentation Online

The following sites provide technical documentation related to Java technology.

`http://developer.sun.com/`

http://java.sun.com/docs/

# We Welcome Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at:

wtk-comments@sun.com

CHAPTER 1

# Introduction

The J2ME Wireless Toolkit provides an emulation environment for the development of MIDP applications. This document provides instructions for customizing the toolkit in two useful ways:

- Creating new emulator skins
- Creating obfuscator plug-ins

The remainder of this chapter briefly describes each of these customizations.

## 1.1 Creating New Emulator Skins

There are three ways to customize the emulators in the J2ME Wireless Toolkit:

1. Download third-party emulators and install them into the J2ME Wireless Toolkit. For details, see Section 4.7, "Using Third Party Emulators," in the *J2ME Wireless Toolkit User's Guide*.
2. Create a new emulator skin based on the J2ME Wireless Toolkit's default emulator. This process is described in Chapter 2, "Skinning the Emulator."
3. Customize the default emulator implementation. To do this you'll want to license the J2ME Wireless Toolkit source code to customize the emulator implementation.

## 1.2 Creating Obfuscator Plug-Ins

An *obfuscator* is a tool that is used to reduce the size of an executable MIDlet suite. Smaller MIDlet suites mean lower download times, which in the current bandwidth-starved wireless world means less waiting, and possibly lower airtime charges, for users.

The J2ME Wireless Toolkit includes support for the ProGuard obfuscator (`http://proguard.sourceforge.net/`), but it includes a flexible architecture that allows for any type of obfuscator.

Chapter 3, "Creating an Obfuscator Plug-in," provides the technical details.

CHAPTER 2

# Skinning the Emulator

This chapter describes how emulator skins are defined. You can modify existing skins or create new skins for the emulator. This process is known as *skinning* the emulator.

## 2.1 The Skin Property File

Emulator skins are defined by a single property file. Each skin property file is contained in its own subdirectory of *{toolkit}*`\wtklib\devices`, where *{toolkit}* is the installation directory of the J2ME Wireless Toolkit. The name of the property file matches the directory name.

For example, the `DefaultColorPhone` skin is defined by `DefaultColorPhone.properties` in the *{toolkit}*`\wtklib\devices\DefaultColorPhone` directory.

The skin property file defines the appearance and behavior of the emulator skin. It includes pointers to images and sounds that may or may not reside in the same directory. For example, the `DefaultColorPhone` directory contains images for the phone itself, but the icons and sounds for `DefaultColorPhone` are defined in `wtklib\devices\Share`.

The remainder of this chapter describes the contents of the skin property file. The property file is a plain text file. You can use any text editor to modify it. In general, entries in the property file have a property name followed by a value. A colon or equals sign separates the name and value. Lines that begin with a hash mark (#) are comments.

The simplest way to create a new skin is to copy an existing one and modify it. For example:

1. Copy the `DefaultColorPhone` directory.
2. Name the new directory with the name of your new skin.

3. Rename the properties file to match the directory name. If you named the directory `NewSkin`, rename its contained property file `NewSkin.properties`.

## 2.2 Skin Appearance

The overall appearance of the emualtor skin is determined by a variety of factors, each of which is described in this section:

- Skin images
- Screen bounds and paintable area
- Screen characteristics
- Icons
- Fonts
- Commands
- Sounds

### 2.2.1 Skin Images

Much of a skin's appearance is determined by three images:

1. The *default* image shows the device in a neutral state.
2. The *highlighted image* shows the device with all the buttons highlighted, as they are when the user moves the mouse over the buttons.
3. The *pressed* image shows the device will all its buttons pressed.

Each of these images shows the entire device. The J2ME Wireless Toolkit uses portions of these images to show button highlights and button presses.

For example, the three images from `DefaultColorPhone` are shown in FIGURE 1.

**FIGURE 1** Images for `DefaultColorPhone`: neutral, highlighted, and pressed

A close-up of the keypad is shown here so you can see the differences in the three images.

**FIGURE 2** Emulator skin image details: neutral, highlighted, and pressed


In the skin property file, the three image files are specified with the following properties:

`default_image=`<*image file name*>
`highlighted_image=`<*image file name*>
`pressed_buttons_image=`<*image file name*>

The image files can be PNG, GIF, or JPEG. They should all be the same dimensions.

For example, `DefaultColorPhone.properties` has the following entries:

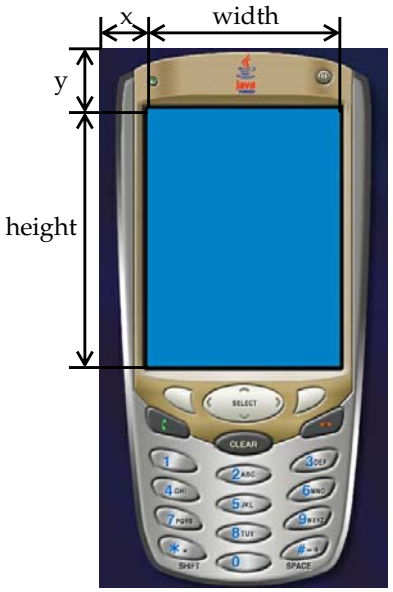
```
default_image=neutral.png
highlighted_image=hilight.png
pressed_buttons_image=pressed.png
```

## 2.2.2 Screen Bounds and Paintable Area

The screen represents the display of a real device. It is defined by the overall screen bounds, the *paintable* bounds, and other parameters that determine factors like the number of colors.

The overall screen bounds are the total area of the display. They are defined in pixel measurements relative to the origin of the image files, which is in the upper left corner.

**FIGURE 3** The bounds of the screen




The screen bounds are specified in the property file as follows:

```
screen.x=<x coordinate>
screen.y=<y coordinate>
screen.width=<width>
screen.height=<height>
```

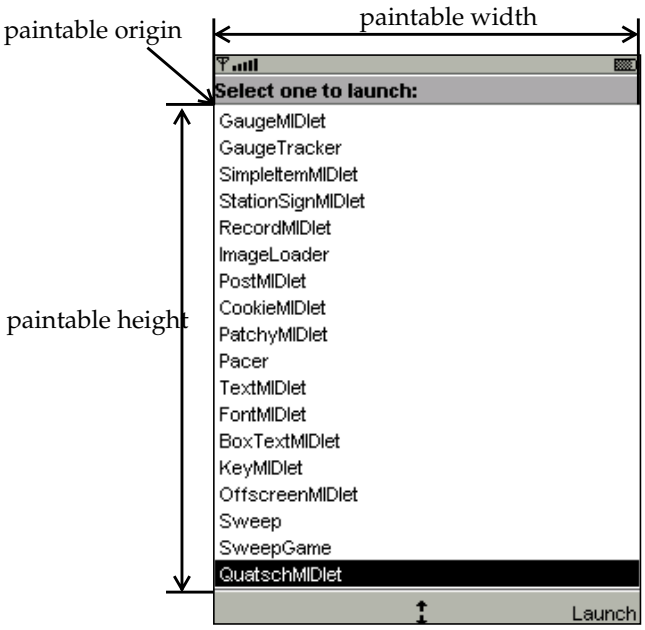
For example:

```
screen.x=60
screen.y=76
screen.width=240
screen.height=320
```

Most devices do not make their full display area available to MIDP applications. The remainder of the screen is generally reserved for icons and indicators of various kinds. Similarly, the J2ME Wireless Toolkit emulator allows you to define a subset of the full screen, called the *paintable* area, that is available for MIDP applications. The origin of the paintable area is expressed in coordinates relative to

the upper left corner of the display. For example, the `DefaultColorPhone` emulator skin uses a top bar for icons and a bottom bar for soft labels and other icons, as shown in FIGURE 4.

**FIGURE 4** The paintable screen area in `DefaultColorPhone`




In the emulator skin property file, the paintable area is expressed as follows:

```
screenPaintableRegion.x=<x coordinate>
screenPaintableRegion.y=<y coordinate>
screenPaintableRegion.width=<width>
screenPaintableRegion.height=<height>
```

For example:

```
screenPaintableRegion.x=0
screenPaintableRegion.y=10
screenPaintableRegion.width=240
screenPaintableRegion.height=290
```

---

**Note –** For full screen mode (in MIDP 2.0), the emulator uses the area beginning at the paintable area origin and extending through the bottom right corner of the screen. In `DefaultColorPhone`, this is the entire screen region with the exception of the top bar.

---

## 2.2.3 Screen Characteristics

The emulator skin property file determines the number of colors supported by the screen and the aspect ratio of the pixels. First, the following property specifies whether the emulator skin uses color or grayscale.

`isColor=`<*true for color or false for grayscale*>

Another property, `colorCount`, specifies the number of available colors. For grayscale devices it specifies the number of gray levels.

`colorCount=`<*number*>

For example, `DefaultColorPhone` has a color screen with 4096 colors:

```
isColor=true
colorCount=0x1000
```

The emulator's handling of alpha (transparency) is determined by the following property:

`enableAlphaChannel=`<*true or false*>

Gamma correction can also be enabled by using the following property:

`gamma=`<*number, where 1 means no error correction*>

Double buffering can be enabled or disabled with the following property:

`screenDoubleBuffer = `<*true or false*>

The background color that is used for the non-paintable areas of the screen is defined as follows:

`screenBorderColor=`<*color*>

For example, `DefaultColorPhone` uses the following color:

```
screenBorderColor=0xb6b6aa
```

On grayscale devices, the background color of the screen can be set using the following property:

`screenBGColor=`<*color*>

## 2.2.4 Icons

The J2ME Wireless Toolkit emulator supports the use of icons, which are small images that convey information to the user. Usually, icons are placed on the display but outside the paintable area. The emulator implements a fixed set of icons which are described in TABLE 1.

**TABLE 1** Emulator icons

| Name | Description |
|---|---|
| battery | Shows battery state |
| domain | Indicates the protection domain of the running MIDlet |
| down | Indicates that scrolling is possible |
| inmode | Indicates the input mode: lower case, upper case, numbers |
| internet | Shows Internet activity |
| left | Indicates that scrolling is possible |
| reception | Shows wireless signal strength |
| right | Indicates that scrolling is possible |
| up | Indicates that scrolling is possible |

Icons are defined with a location (measured relative to the origin of the screen), a default state, and a list of images that correspond to the possible states. For example, here is the definition of the `down` icon in `DefaultColorPhone`. This icon is a downward-pointing arrow that appears when a list or form is shown that is taller than the available screen space.

```
icon.down: 113, 314, off
icon.down.off:
icon.down.on: ../Share/down.gif
```

The first line specifies the location where the icon will be shown, which for `DefaultColorPhone` is a location in the center of the bottom bar, outside the paintable screen area. The default state is `off`.

There is no image file that corresponds to the `off` state, but the `on` state uses the image `down.gif` from the `wtklib\devices\Share` directory.

Another interesting example is the `inmode` icon, which includes seven states with six corresponding image files:

```
icon.inmode: 113, 2, off
icon.inmode.off:
icon.inmode.ABC: ../Share/ABC.gif
icon.inmode.abc: ../Share/abc_lower.gif
icon.inmode.123: ../Share/123.gif
icon.inmode.kana: ../Share/kana.gif
icon.inmode.hira: ../Share/hira.gif
icon.inmode.sym: ../Share/sym.gif
```

Another aspect of the emulator that is similar to an icon is the network indicator. Instead of being located in the screen, the network indicator is shown on the emulator skin. In `DefaultColorPhone`, the network indicator is shown as a small green light in the upper left of the emulator skin. The network indicator is defined using two properties:

`netindicator.image:` *<image>*
`netindicator.bounds:` *<x>*`,` *<y>*`,` *<width>*`,` *<height>*

For example, in `DefaultColorPhone`, the network indicator looks like this:

```
netindicator.image: net_indicator.png
netindicator.bounds: 53, 27, 30, 30
```

The width and height should match the width and height of the network indicator image.

## 2.2.5 Fonts

The fonts used by the emulator are defined in the skin property file. In essence you can define a font for each of the faces, styles, and sizes that are available in MIDP's `Font` class. The format is as follows:

`font.`*<face>*`.`*<style>*`.`*<size>*`:` *<font specifier>*

You can surmise the fact, style, and size parameters from the MIDP `Font` API, except the identifiers are lower case in the emulator skin property file. The font face is `system`, `monospace`, or `proportional`, the style is `plain`, `bold`, or `italic`, and the size is `small`, `medium`, or `large`.

The font specifier follows the convention laid out in the J2SE `java.awt.Font` class. The following example from `DefaultColorPhone` defines the proportional italic fonts in all three sizes:

```
font.proportional.italic.small: SansSerif-italic-9
font.proportional.italic.medium: SansSerif-italic-11
font.proportional.italic.large: SansSerif-italic-14
```

You need to specify a default font that will be used in case no other definition is available. In `DefaultColorPhone`, a 10-point SansSerif font is used for the default:

```
font.default=SansSerif-plain-10
```

Fonts may also be underlined. By default, this is supported by the MIDP implementation, but you can disable for specific fonts like this:

`font.`*<face>*`.`*<style>*`.`*<size>*`.underline.enabled=false`

If you wish, you can disable underlining for all fonts like this:

```
font.all.underline.enabled=false
```

Instead of using system fonts, you have an additional option of using a *bitmap* font. A bitmap font is simply an image that contains character shapes for a font. The bitmap font image is a single line of text containing one of each character shape. To define a bitmap font, use the following property:

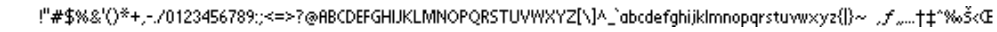
`font.`*<name>*`=`*<font property file>*

The font property file contains the following property definitions:

```
font_image = <image file>
font_height = <font height>
font_ascent = <font ascent>
font_descent = <font descent>
font_leading = <font leading>
```

The image file should be in PNG, GIF or JPEG format. It should contain a row of characters:

**FIGURE 5** Bitmap font image




The height, ascent, descent, and leading are all specified in pixels. If you are unfamiliar with these font terms, refer to the J2SE documentation for `java.awt.FontMetrics`.

The font property file should also contain a list of mappings between ASCII character codes and horizontal pixel offsets into the image. In the following example, the ASCII code 65 is mapped to the horizontal offset 124:

```
ascii_x-65=124
```

Once a bitmap font is defined, its name may be used as a font specifier.

## 2.2.6 Soft Button Labels

Soft buttons are buttons without a fixed function. They will be fully discussed later in this chapter. Labels for the soft buttons are shown on the screen. The emulator skin property file determines where and how the soft button labels are shown.

The fonts for the soft button labels are defined using font aliases, which are short names that you assign to a font. Each soft button label is described by a property:

`softbutton.`*<n>*`=`*<x>*`, `*<y>*`, `*<width>*`, `*<height>*`, `*<font alias>*`, `*<alignment>*

Valid values for *alignment* are `left`, `right`, and `center`.

For example, the following properties tell the toolkit to use a Courier 12-point font for the soft button labels. First the font alias `softButton` is defined. The first label is left-justified, while the second is right-justified.

```
font.softButton=Courier-plain-12
softbutton.0=1,306,78,16, softButton, left
softbutton.1=160,306,78,16, softButton, right
```

### 2.2.7 Sounds

MIDP alerts have associated sounds. In the J2ME Wireless Toolkit emulator, sounds are defined using files, one for each type enumerated in the MIDP `AlertType` class. The emulator can use any sound file type that is supported by the underlying J2SE implementation. In J2SE SDK 1.4, this includes AIFF, AU, WAV, MIDI, and RMF. For example, here are the definitions in `DefaultColorPhone`:

```
alert.alarm.sound:   ../Share/mid_alarm.wav
alert.info.sound:    ../Share/mid_info.wav
alert.warning.sound: ../Share/mid_warn.wav
alert.error.sound:   ../Share/mid_err.wav
alert.confirmation.sound: ../Share/mid_confirm.wav
```

A *default* sound will be played if no sound is defined for a specific alert type:

`alert.confirmation.sound: `*<sound file>*

In addition, you can define a sound that will be played to simulate a phone's vibration. In `DefaultColorPhone`, it looks like this:

```
vibrator.sound: ../Share/vibrate.wav
```

## 2.3 Mapping User Input

There are two parts to describing an emulator skin. The first part is the appearance, which is described above. The second part defines how user input is mapped in the emulator.

### 2.3.1 The Keyboard Handler

A *keyboard handler* takes button presses and performs an appropriate action in the emulator. For example, if you use the mouse to press one of the soft buttons, it is the keyboard handler that makes the appropriate action happen in the emulator.

The keyboard handler defines a set of standard button names, which you will use when you define buttons. You just have to tell the emulator where the buttons are located in the skin and the keyboard handler takes care of the rest.

The J2ME Wireless Toolkit emulator includes two keyboard handlers, one for phone devices with an ITU-T keypad (`DefaultKeyboardHandler`) and one for devices with a full Qwerty keyboard. For example, `DefaultColorPhone` includes this keyboard handler property:

```
keyboard.handler = com.sun.kvem.midp.DefaultKeyboardHandler
```

`DefaultKeyboardHandler` recognizes the following standard button names: `0` through `9`, `POUND`, `ASTERISK`, `POWER`, `SEND`, `END`, `LEFT`, `RIGHT`, `UP`, `DOWN`, `SELECT`, `SOFT1`, `SOFT2`, `SOFT3`, `SOFT4`, `USER1` through `USER10`.

In `QwertyDevice`, the keyboard handler looks like this:

```
keyboard.handler = com.sun.kvem.midp.QwertyKeyboardHandler
```

`QwertyKeyboardHandler` supports the same buttons as `DefaultKeyboardHandler` and also includes buttons found on a standard keyboard like alphabetic keys, shift, and alt.

### 2.3.2 Buttons

Buttons are defined using a name and a set of coordinates. If two sets of coordinates are supplied, a rectangular button is defined. If more than two sets of coordinates are present, a polygonal area is used for the button.

The button region is defined relative to the device skin image. When the user moves the mouse over a defined button region, the corresponding region from the skin highlight image is shown. If the user presses a button, the corresponding region from the skin pressed image is shown.

By themselves, buttons aren't very interesting. They just associate a button name with a rectangular or polygonal region. It's the keyboard handler's job to map the button name to a function in the emulator. Later, you'll see how keys on your desktop computer's keyboard can be mapped to buttons.

The following property shows how to define a rectangular region for the 5 button. Its origin is `140, 553`, with a width of `84` and a height of `37`.

```
button.5 = 140, 553,  84,  37
```

Here is an example polygonal definition for the asterisk button:

```
button.ASTERISK = 66, 605, 110, 606, 140, 636, 120, 647, 70, 637
```

This polygon is defined using straight line segments connecting the listed points:

```
66, 105
110, 606
140, 636
120, 647
70, 637
```

## 2.3.3 Assigning Desktop Keyboard Keys to Buttons

Buttons can have one or more associated desktop keyboard keys. This means that you can use your desktop keyboard to control the emulator instead of having to move the mouse over on the device skin and press the mouse button.

For example, `DefaultColorPhone` allows you to press F1 on your desktop keyboard to simulate the left soft button. The left soft button is defined as `SOFT1` in the property file:

```
button.SOFT1 = 78, 417, 120, 423, 126, 465, 74, 440
```

And the desktop keyboard shortcut is defined thus:

```
key.SOFT1 = VK_F1
```

The actual key definitions are *virtual key codes*, which are defined in J2SE's `java.awt.event.KeyEvent` class. See the J2SE documentation for details.

You can assign multiple desktop keyboard keys to a button, if you wish. In the following example from `DefaultColorPhone`, the `5` key or the the number pad `5` key on your desktop keyboard are both defined as shortcuts for the `5` button on the emulator skin:

```
key.5 = VK_5 VK_NUMPAD5
```

## 2.3.4 Mapping Game Keys

Game actions are already defined in `DefaultKeyboardHandler`, but you can specify your own game actions with `QwertyKeyboardHandler`. Use lines of the form:

`game.`*<function>* `=` *<button name>*

The function can be one of `LEFT`, `RIGHT`, `UP`, `DOWN` and `SELECT`. Standard button names are described earlier in this chapter.

The default settings are:

```
game.UP = UP
game.DOWN = DOWN
game.LEFT = LEFT
game.RIGHT = RIGHT
game.SELECT = SELECT
```

## 2.3.5 Mapping Keys to Characters

With `QwertyKeyboardHandler`, you can specify which character is generated by a button press either alone or in combination with the shift or alt buttons.

Use a line of the form:

`keyboard.handler.qwerty.`*<button>* `= '`*<base character>*`' '`*<shift character>*`' '`*<alternate character>*`'`

The base character is the character the button normally generates, shift character is the character used when the button is pressed at the same time as shift, and alternate character is the character generated when the button is pressed at the same time as alt.

There are two ways you can do a button press at the same time as pressing shift or alt:

- Map the buttons to the keyboard, as in the previous section, and press the key associated with the button at the same time as the shift or alt keys.
- Press the button shift-lock or alt-lock and then do the button press. Press shift-lock or alt-lock again to revert to the initial state.

For example:

```
keyboard.handler.qwerty.A = 'a' 'A' '?'
```

## 2.3.6 Mapping Commands to Soft Buttons

*Commands* are part of the MIDP specification. They are a flexible way to specify actions that should be available to the user, without mandating how a particular device makes them available.

In general, MIDP devices use soft buttons to invoke commands. The command text is shown on the display, somwhere physically near to the soft buttons. If there are more commands than available soft buttons, the implementation will show one soft button label as a menu. Pressing the menu soft button brings up a menu of available commands.

The J2ME Wireless Toolkit emulator allows you to specify where you want certain types of commands to appear, based on the command types specified in `javax.microedition.lcdui.Command`. For example, on an emulator skin with two soft buttons, you might prefer that `BACK` and `EXIT` commands always appear on the left soft button, while `OK` commands should appear on the right soft button.

You can specify these types of preferences in the emulator skin property file, using lines like the following:

`command.keys.`*<command type>*`=`*<button>*

For example, `DefaultColorPhone` defines command preferences this way:

```
command.keys.BACK = SOFT1
command.keys.EXIT = SOFT1
command.keys.CANCEL = SOFT1
command.keys.STOP = SOFT1

command.keys.OK = SOFT2
command.keys.SCREEN = SOFT2
command.keys.ITEM = SOFT2
command.keys.HELP = SOFT2
```

By specifying additional button names, you can specify other preferred buttons for a particular command type. For example, this line tells the emulator that `BACK` commands should be mapped to `END`, if it is available, or `SOFT1` otherwise.

```
command.keys.BACK = END SOFT1
```

Finally, if you wish, you can specify that a soft button will only be used for specific command types. The following definition allows only the command types `BACK`, `EXIT`, `CANCEL`, and `STOP` to be mapped to the `SOFT1` key.

```
command.exclusive.SOFT1 = BACK EXIT CANCEL STOP
```

## 2.3.7 The Command Menu

When there are more commands than available soft buttons, commands are placed in a menu. The J2ME Wireless Toolkit emulator offers control over the command menu. You can choose the button which is used to show the menu, the buttons that are used to traverse the items in the menu, and the text labels that are shown for the menu.

The following property, from `DefaultColorPhone`, tells the emulator skin to use the second soft button to show or hide the menu.

```
command.menu.activate = SOFT2
```

By default, the `UP` and `DOWN` buttons are used to traverse the menu, while `SELECT` is used to choose a command. You can change these assignments using the following properties:

```
command.menu.select = <button>
command.menu.up = <button>
command.menu.down = <button>
```

### 2.3.8 Pausing and Resuming

The MIDP specification allows applications (`MIDlets`) to be paused at any time, possibly in response to other phone events like incoming calls.

You can use the emualtor skin property file to define desktop keyboard shortcuts for pausing and resuming MIDlets. `DefaultColorPhone`, for example, uses F6 for pausing (suspending) and F7 for resuming:

```
midlet.SUSPEND_ALL = VK_F6
midlet.RESUME_ALL = VK_F7
```

### 2.3.9 Pointer Events

A single property determines whether the emulator skin has a touch screen. If it does, pointer events will be delivered to `Canvas`es.

```
touch_screen=<true or false>
```

## 2.4 Locale and Character Encoding

A locale is a geographic or political region or community that shares the same language, customs, or cultural convention. In software, a locale is a collection of files, data, and code, which contains the information necessary to adapt software to a specific geographical location.

Some operations are locale-sensitive and require a specified locale to tailor information for users, such as:

- Messages displayed to the user
- Cultural information such as, dates and currency formats

In the J2ME Wireless Toolkit emulator, the default locale is determined by the platform's locale.

To define a specific locale, use the following definition:

`microedition.locale:` *<locale name>*

A locale name is comprised of two parts separated by an dash (-), for example, `en-US` is the locale designation for English/United States while `en-AU` is the designation for English/Australia.

The first part is a valid ISO Language Code. These codes are the lower-case two-letter codes defined by ISO-639. You can find a full list of these codes at a number of sites, such as:

`http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt`

The second part is a valid ISO Country Code. These codes are the upper-case two-letter codes defined by ISO-3166. You can find a full list of these codes at a number of sites, such as:

`http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html`

The input and output APIs in CLDC use named character encodings to convert between 8-bit characters and 16-bit Unicode characters. Specific MIDP implementation might make only a small set of encodings available for MIDlets to use.

In the emulator, the default encoding is default encoder of the platform you are running on. Your emulator might use other encodings, such as UTF-8 and UTF-16, providing they are available in the J2SE platform.

To define the character encoding used by an emulator skin, use the following definition:

`microedition.encoding:` *<encoding>*

To define the set of all available encodings, use the following definition:

`microedition.encoding.supported:` *<list of encodings>*

For example:

```
microedition.encoding: UTF-8
microedition.encoding.supported: UTF-8, UTF-16, ISO-8859-1,
   ISO-8859-2, Shift_JIS
```

To support all encodings supported by the J2SE platform, leave the `microedition.encoding.supported` definition blank, as in:

```
microedition.encoding.supported:
```

---

**Note –** Note – The encoding ISO-8859-1 is always available to applications running on emulated devices, whether or not it is listed in the `microedition.encoding.supported` entry.

---

CHAPTER 3

# Creating an Obfuscator Plug-in

The J2ME Wireless Toolkit allows you to use a bytecode obfuscator to reduce the size of your MIDlet suite JAR. The toolkit comes with support for ProGuard and RetroGuard, as described in the *J2ME Wireless Toolkit User's Guide*.

If you want to use a different obfuscator, you can write a plug-in for the J2ME Wireless Toolkit.

## 3.1 Writing the Plug-in

Obfuscator plug-ins extend the `com.sun.kvem.environment.Obfusctor` inteface. The interface itself is contained in *{toolkit}*`\wtklib\kenv.zip`.

The `Obfuscator` interface contains two methods that you must implement:

```
public void createScriptFile(File jadFilename, File projectDir);

public void run(File jarFileObfuscated, String wtkBinDir,
    String wtkLibDir, String jarFilename, String projectDir,
    String classPath, String emptyAPI) throws IOException;
```

To compile your obfuscator plug-in, make sure to add `kenv.zip` to your `CLASSPATH`.

For example, here is the source code for a very simple plug-in. It doesn't actually invoke an obfuscator, but it shows how to implement the `Obfuscator` interface.

```
import java.io.*;


public class NullObfuscator
    implements com.sun.kvem.environment.Obfuscator {
  public void createScriptFile(File jadFilename, File projectDir) {
    System.out.println("NullObfuscator: createScriptFile()");
  }

  public void run(File jarFileObfuscated, String wtkBinDir,
      String wtkLibDir, String jarFilename, String projectDir,
      String classPath, String emptyAPI) throws IOException {
    System.out.println("NullObfuscator: run()");
  }
}
```

Suppose you save this as *{toolkit}*`\wtklib\test\NullObfuscator.java`. Then you can compile it at the command line like this:

```
set classpath=%classpath%;\WTK22\wtklib\kenv.zip
javac NullObfuscator.java
```

## 3.2 Configuring the Toolkit

Once you've written an obfuscator plug-in, you have to tell the toolkit where to find it. To do this, edit *{toolkit}*`\wtklib\Windows\ktools.properties`. You'll need to edit the obfuscator plug-in class name and tell the toolkit where to find the class. If you're following along with the example, edit the properties as follows:

```
obfuscator.runner.class.name: NullObfuscator
obfuscator.runner.classpath: wtklib\\test
```

Restart KToolbar and open a project. Now choose **Project > Package > Create Obfuscated Package**. In the KToolbar console, you'll see the output of `NullObfuscator`:

```
Project settings saved
Building "Tiny"
NullObfuscator: createScriptFile()
NullObfuscator: run()
Wrote C:\WTK22\apps\Tiny\bin\Tiny.jar
Wrote C:\WTK22\apps\Tiny\bin\Tiny.jad
Build complete
```

# Index

## P



## Q



## S



## T