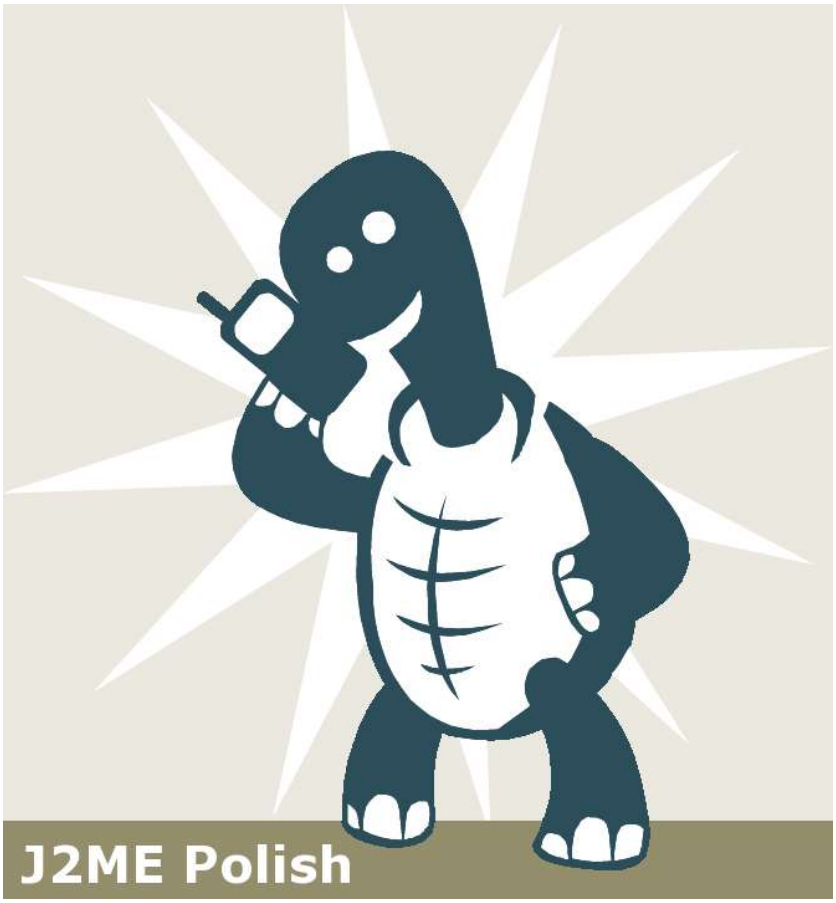
The Complete Guide to J2ME Polish

The Solution for
Professional MIDlet Programming



ENOUGH SOFTWARE

Table of Contents

Introduction

J2ME developers face a serious problem: either they use only the common denominator of J2ME devices – the MIDP/1.0 standard – or they choose to support only a few handsets, for which they optimize their applications with a lot of energy. The open source tool J2ME Polish promises a solution to this problem – and more.

J2ME Polish eases the development of mobile applications significantly. The optimization for the different handsets is done automatically; developers can make use of the device specific capabilities without endangering the compatibility of the application. Thanks to the optional GUI the application can be designed using the Web standard CSS. Web designers can now design J2ME applications with their know-how and without changing the source code of an application.

The J2ME Polish framework is divided into the J2ME and the build section. In the J2ME section you will find some utilities like an ArrayList as well as the already mentioned GUI API, which is compatible to the MIDP/1.0 and MIDP/2.0 standard. In the build section all steps for building a J2ME application are done: preprocessing and device-optimization, compilation, preverification, creation of jar- and jad-files and obfuscation of the application. The build process is based on Ant – the standard tool for building Java applications. J2ME Polish thus works with all IDEs. In the build area there is an additional device database, which is based on XML and can be extended easily. The device database is the key to the automatic optimization of the applications.

Since J2ME Polish is compatible to the MIDP/1.0 and MIDP/2.0 standard, the source code of existing applications does not need to be changed. Even existing preprocessing directives can be used, since J2ME Polish supports all directives of the antenna-preprocessor. The programmer only needs to create a build.xml file which controls the build process. When the extended capabilities of the J2ME Polish GUI should be used, one should (but is not required to) add some style-directives in the code. Adjustments to specific devices are easy with the help of the preprocessing and the device database.

Obviously one can extend and use the more advanced features of the J2ME Polish framework. You can use your own widgets, add your own preprocessor, or integrate your own obfuscator with ease.

I hope you find J2ME Polish useful and I wish you a lot of fun with it!

Bremen. Germany, June 2004

Robert Virkus / Enough Software

Installation

Requirements

To use J2ME Polish, following components need to be installed:

- Java 2 Standard Edition SDK 1.4 or higher, `http://java.sun.com/j2se/1.4.2/index.jsp`
- Java Wireless Toolkit, `http://java.sun.com/products/j2mewtoolkit/index.html`
- Favorite IDE, for example Eclipse 3.0, `http://www.eclipse.org`
- Ant 1.5 or higher, if not already integrated in the IDE, `http://ant.apache.org`

Installation of J2ME Polish

For installing J2ME Polish download the latest release and start the installation either by double-clicking the file or by calling "java -jar j2mepolish-$VERSION.jar" from the command-line (where $VERSION denotes the current version of J2ME Polish). It is recommended to install J2ME Polish into a project-folder of your workspace, e.g. "$HOME/workspace/j2mepolish".

The installer of J2ME Polish contains a sample MIDlet application with two different designs. This application is optimized for Nokia Series 60 devices, so it is recommended to have such an emulator installed. The Nokia emulator can be retrieved from http://forum.nokia.com.

To build the sample application, run the included build.xml from your favorite Java IDE or call "ant" from the command-line. Call "ant notest j2mepolish" to build and obfuscate the sample application.

Updating J2ME Polish

When you want to update an existing installation, you do not need to re-install the whole package. For saving bandwidth you can download just the update-package. This package just containts the two JAR-files enough-j2mepolish-build.jar and enough-j2mepolish-client.jar which need to be copied into the "import"-folder of your installation.

Integrating J2ME Polish into Eclipse

To integrate J2ME Polish into Eclipse it is best to install J2ME Polish (with the included sample application) into a new folder in your workspace called "myproject" (or any other name). Then start Eclipse and create a new project called "myproject". Eclipse then automatically integrates all source-files and sets the classpath.

Optionally you can change the build-settings by modifying the file "build.xml" (which is located in the root of your project). For example you can change the deviceRequirements if you want to. The example is optimized to Nokia Series 60 devices.

You can now create the JAR and JAD files by right-clicking the build.xml file, selecting "Run Ant" and running Ant in the next dialog. You will find the JAR and JAD files then in the "dist" folder of the project. If you want to access them from within Eclipse, you might need to refresh your project: Right-click the project and select "Refresh".

After you have installed J2ME Polish, you will find following structure in your project (assuming

that your project is called “myproject” and your workspace “workspace”):

Folder / File	***Description***
workspace/myproject	The applications project
workspace/myproject/build.xml	The controller of the build process
workspace/myproject/devices.xml	Definition of J2ME devices (only installed when you selected to install the external device database)
workspace/myproject/vendors.xml	Definition of J2ME device vendors (only installed when you selected to install the external device database)
workspace/myproject/groups.xml	Definition of device groups (only installed when you selected to install the external device database)
workspace/myproject/apis.xml	Definition of some common J2ME libraries (only installed when you selected to install the external device database)
workspace/myproject/resources	Folder for all resources and design descriptions of the project
workspace/myproject/resources/polish.css	Basic design description
workspace/myproject/import	Needed APIs
workspace/myproject/build	Temporary build folder, will be created automatically. Should not be shared in CVS and similar systems.
workspace/myproject/dist	Folder for the ready-to-deploy applications. It will be created automatically. Should not be shared in CVS and similar systems.

Tip: You don't need the files “devices.xml”, “vendors.xml”, “groups.xml” and “apis.xml”, since they are also contained in the jar-file “enough-j2mepolish-build.jar”. If you don't want to change them, you can delete them.

The Device Database

All J2ME devices are defined in the file “devices.xml”. Every device has a vendor, which is defined in the file “vendors.xml”. Every device can belong to an arbitrary number of groups, which are defined in the file “groups.xml”. Libraries can be managed using the file “apis.xml”.

All these files are contained in the file “enough-j2mepolish-build.jar”, so when you do not find them, extract them from the jar file into the root folder of the project.

vendors.xml

The vendors of J2ME devices are defined in vendors.xml. An example definition is:

```
<vendors>
	<vendor>
		<name>Nokia</name>
		<features></features>
		<capability name="colors.focus" value="0x5555DD" />
	</vendor>
	<vendor>
		<name>Siemens</name>
	</vendor>
</vendors>
```

In this code 2 vendors are defined – Nokia and Siemens. Vendors can possess features and capabilities, which are inherited by all devices of that vendor. These abilities can be used during the proprocessing (compare chapter “The world of preprocessing”, page 26). Features are just a comma separated list, whereas capabilities always have a name and a value.

groups.xml

The device groups are defined in the file groups.xml. A device can be a member in any number of groups.

```
<groups>
	<group>
		<name>Nokia-UI</name>
		<features></features>
		<capability name="classes.fullscreen"
			value="com.nokia.mid.ui.FullCanvas" />
		<capability name="JavaPackage" value="nokia-ui" />
	</group>
	<group>
		<name>Series60</name>
		<parent>Nokia-UI</parent>
		<capability name="JavaPlatform"
			value="MIDP/1.0" />
	</group>
</groups>
```

The group “Nokia-UI” is used for devices which support the UI API of Nokia. The group “Serie60” extends this group. Any capabilities or features defined in the groups can be overridden or completed by the device definitions.

devices.xml

In devices.xml all J2ME devices are defined.

```
<devices>
	<device>
		<identifier>Motorola/i730</identifier>
		<user-agent>MOTi730</user-agent>
		<capability name="SoftwarePlatform.JavaPlatform"
			value="MIDP/2.0" />
		<capability name="HardwarePlatform.ScreenSize"
			value="130x130" />
		<capability name="HardwarePlatform.BitsPerPixel"
			value="16" />
		<capability name="HardwarePlatform.CameraResolution"
			value="300x200" />
		<capability name="SoftwarePlatform.JavaProtocol"
			value="UDP, TCP, SSL, HTTP, HTTPS, Serial"/>
		<capability name="SoftwarePlatform.JavaPackage"
			value="wmapi, mmapi, motorola-lwt, location-api"/>
		<capability name="SoftwarePlatform.HeapSize"
			value="1.15 MB" />
		<capability name="SoftwarePlatform.MaxJarSize"
			value="500 kb" />
	</device>
	<device>
		<identifier>Nokia/6600</identifier>
		<user-agent>Nokia6600</user-agent>
		<groups>Series60</groups>
		<features>hasCamera</features>
		<capability name="ScreenSize" value="176x208"/>
		<capability name="BitsPerPixel" value="16"/>
		<capability name="JavaPackage" value="mmapi, btapi, wmapi" />
		<capability name="JavaPlatform" value="MIDP/2.0" />
	</device>
</devices>
```

In the example the Motorola Ident 730 and the Nokia 6600 are defined. Capabilities can be grouped into hardware- and software-capabilities. This grouping is optional and will not be distinguished in the preprocessing step, thus you cannot use the same name for a SoftwarePlatform and a HardwarePlatform capability.

The identifier consists of the vendor and device-name. The structure is “[vendor]/[device-name]”.
The <features>-element defines preprocessing symbols. These can be checked in the source code for example with “//#ifdef [symbol-name]“. Several features need to be separated by comma:
`<features>supportsPointer, roundKnobs</features>`
Groups can be defined explicitly with the <groups>-element. All group-names to which the device belongs are in a comma separated list:
`<groups>Series60, SomeGroup</groups>` defines 2 groups for the device. Groups can also be defined indirectly by the capabilities of the device.
The <device>-Element supports the single attribute “supportsPolishGui”:
`<device supportsPolishGui="true">...`
Normally it is calculated whether a device supports the J2ME Polish GUI: when it has a color depth of at least 8 bits per pixel and a heap size of 500 kb or more, the device supports the GUI. The “supportsPolishGui”-attribute overrides any calculations.

Capabilities

Capabilities can be defined in two different ways:

```
<capability name="ScreenSize" value="176x208"/>
```
is equivalent with
```
<capability>
      <capability-name>ScreenSize</capability-name>
      <capability-value>176x208</capability-value>
</capability>
```
You can use capabilities with any name, but following categories are predefined:

Capability	*Explanation*	*Preprocessing-Access*	*Groups*
BitsPerPixel	Color depth: 1 is monochrome, 4 are 16 colors, 8 = 256 colors, 16 = 65.536 colors, 24 = 16.777.216 colors	*Variable:* polish.BitsPerPixel, *Symbols:* polish.BitsPerPixel.1 or polish.BitsPerPixel.4, polish.BitsPerPixel.16 etc	At 8 bits per pixel for example: BitsPerPixel.4+ and BitsPerPixel.8
ScreenSize	Width times height of the screen resolution in pixels, e.g. “176 x 208”	*Variables:* polish.ScreenSize, polish.ScreenWidth, polish.ScreenHeight *Symbols (example):* polish.ScreenSize.176x208 polish.ScreenWidth.176 polish.ScreenHeight.208	-
CanvasSize	Width times height of an MIDP-Canvas.	Like ScreenSize	-
CameraResolution	Resolution of the camera.	*Variables:* polish.CameraResolution, polish.CameraWidth, polish.CameraHeight *Symbols (example):* polish.CameraResolution.320x200 polish.CameraWidth.320 polish.CameraHeight.200	-
JavaPlatform	The supported Java platform. Currently either MIDP/1.0 or MIDP/2.0.	*Variable:* polish.JavaPlatform *Symbols:* polish.midp1 or polish.midp2	midp1 or midp2
JavaPackage	Supported APIs, e.g. “nokia-ui, mmapi”	*Variables:* polish.api, polish.JavaPackage *Symbols (examples):* polish.api.nokia-ui polish.api.mmapi etc	Respectively the name of the supported API, e.g. nokia-ui or mmapi (one group for each supported API).

Capability	Explanation	Preprocessing-Access	Groups
JavaProtocol	Supported data exchange protocols. All MIDP/1.0 devices support the HTTP protocol. All MIDP/2.0 devices support additionally the HTTPS protocol.	*Variable:* polish.JavaProtocol *Symbols (example):* polish.JavaProtocol.serial polish.JavaProtocol.https	-
HeapSize	The maximum heap size, e.g. “500 kb” or “1.2 MB”	*Variable:* polish.HeapSize	-
MaxJarSize	The maximum size of the MIDlet-JAR-bundle, e.g. “100 kb” or “2 MB”	*Variable:* polish.MaxJarSize	-
OS	The operating system of the device, e.g. “Symbian OS 6.1”	*Variable:* polish.OS	-
VideoFormat	The supported video formats of the device, e.g. “3GPP, MPEG-4”	*Variable:* polish.VideoFormat *Symbols (examples):* polish.VideoFormat.3gpp polish.VideoFormat.mpeg-4	-
SoundFormat	The sound formats which are supported by the device, e.g. “midi, wav”	*Variable:* polish.SoundFormat *Symbols (examples):* polish.SoundFormat.midi polish.SoundFormat.wav	-

apis.xml

The file apis.xml defines some common libraries. You do not need to add every API you or your device supports, but if a API is known under several names, it is advised to add that API to apis.xml.

```
<apis>
      <api>
            <name>Nokia User Interface API</name>
            <description>The Nokia User Interface API provides
                  some advanced drawing functionalities.
            </description>
            <names>nokia-ui,nokiaui, nokiauiapi</names>
            <files>nokia-ui.jar, nokiaui.zip</files>
            <path>import/nokia-ui.jar</path>
      </api>
      <api>
            <name>Bluetooth API</name >
            <description>The Bluetooth API provides
                  functionalities for data exchange with other bluetooth devices.
            </description>
            <names>btapi,bt-api, bluetooth, bluetooth-api</names>
            <files>j2me-btapi.jar, bluetooth.zip</files>
            <path>import/j2me-btapi.jar</path>
      </api>
</apis>
```

In the above example two libraries are defined. The <name> element describes the default name of a library, whereas the <names> element defines other possible names of the library. The <files> element defines the names of the library on the filesystem. The <path> element just defines the default path. When that path is not found, J2ME Polish tries to find the API with the help of the file-names defined in the <files> element.

The Build Process

Introduction

The build process creates “ready to deploy”, optimized application-bundles from the source code. The process is controlled by the build.xml file, which is situated at the root of the project. This file is a standard Ant file which is used to control the J2ME Polish-task. You can find a short introduction to Ant in the appendix on page 67. The J2ME Polish-task is separated into the sections “info”, “deviceRequirements” and “build”. During the build following steps are accomplished:

- Selection of the supported devices
- Assembling of the resources
- Preprocessing of the source code, optimizing for the device
- Compilation of the application
- Obfuscation and shrinking of the compiled code
- Preverification
- Creation of the JAR and JAD files

} for each selected device

Definition of the J2ME Polish-task

You need to define the J2ME Polish-task in the build.xml file:

```
<taskdef name="j2mepolish" classname="de.enough.polish.ant.PolishTask"
classpath="import/enough-j2mepolish-
build.jar:import/jdom.jar:import/proguard.jar"/>
```

Now you can use the 2ME Polish-Task under the name “j2mepolish”.

You can also define where to find optional and MIDP APIs. You just need to define an Ant-property with the name “polish.api.[api-name]” and the location of the API:

```
<property name="polish.api.nokia-ui"
location="/home/enough/Nokia/Devices/Series_60_MIDP/lib/ext/nokiaui.zip"/>
```

When you do not define the location of an API, it will be searched in the import folder under the name of the API: import/[api-name].jar or import/[api-name].zip. The Nokia UI API would be searched at workspace/application/import/nokia-ui.jar or at workspace/application/import/nokia-ui.zip.

Example

The following example shows a complete build.xml file:

```
<project
      name="my-j2me-project"
      default="j2mepolish">

<!-- task definition -->
<taskdef name="j2mepolish" classname="de.enough.polish.ant.PolishTask"
classpath="import/enough-j2mepolish-
build.jar:import/jdom.jar:import/proguard.jar:import/retroguard.jar"/>
```

```
<!-- optional J2ME apis -->
<property name="polish.api.mmapi"
 location="/home/user/Nokia/Devices/Series_60/lib/ext/mma.zip" />
<property name="polish.api.wmapi"
 location="/home/user/Nokia/Devices/Series_60/lib/ext/wma.zip" />
<property name="polish.api.nokia-ui"
 location="/home/user/Nokia/Devices/Series_60/lib/ext/nokiaui.zip" />
<property name="polish.api.btapi"
 location="/home/user/Nokia/Devices/Series_60/lib/ext/bluetooth.zip" />


<!-- build targets, each target can be called via "ant [name]",
       e.g. "ant clean", "ant notest j2mepolish" or just "ant" for calling the
default-target -->

<target name="notest" >
      <property name="test" value="false" />
</target>

<target name="init">
      <property name="test" value="true" />
</target>

<target name="j2mepolish"
        depends="init">
      <j2mepolish>
            <!-- general settings -->
            <info
                license="GPL"
                name="SimpleMenu"
                version="1.3.4"
                description="A test project"
                vendorName="enough software"
                infoUrl="http://www.enough.de/dictionary"
                icon="dot.png"
                jarName="${polish.vendor}-${polish.name}-example.jar"
                jarUrl="${polish.jarName}"
                copyright="Copyright 2004 enough software. All rights reserved."
                deleteConfirm="Do you really want to kill me?"
                />

            <!-- selection of supported devices -->
            <deviceRequirements if="test">
                <requirement name="Identifier" value="Nokia/6600" />
            </deviceRequirements>
            <deviceRequirements unless="test">
                <requirement name="JavaPackage" value="nokia-ui" />
                <requirement name="BitsPerPixel" value="4+" />
            </deviceRequirements>

            <!-- build settings -->
            <build
                symbols="showSplash, AnotherExampleSymbol"
                imageLoadStrategy="background"
                fullscreen="menu"
                usePolishGui="true"
                preverify="/home/user/WTK2.1/bin/preverify"
                >
                <!-- midlets definition -->
```

```
                <midlet class="MenuMidlet" name="Example" />
                <!-- project-wide variables - used for preprocessing  -->
                <variables>
                  <variable name="update-url"
                            value="http://www.enough.de/update" />
                </variables>
                <!-- obfuscator settings: do not obfuscate when test is true -->
                <obfuscator unless="test" enable="true" name="ProGuard" />
                <!-- debug settings: only debug when test is true -->
                <debug if="test"
                       enable="true" visual="false" verbose="true" level="error">
                  <filter pattern="de.enough.polish.dict.*" level="debug" />
                  <filter pattern="de.enough.polish.ui.*" level="warn" />
                </debug>
            </build>
      </j2mepolish>
</target>


<target name="clean">
      <delete dir="build" />
      <delete dir="dist" />
</target>
</project>
```

In the first section the J2ME Polish-task and the locations of some optional APIs are defined, followed by the build-targets “notest”, “init”, “j2mepolish” and “clean”. The targets “notest” and “init” are responsible for entering the test mode.

If you call Ant without any arguments, the Ant-property “test” will be set to true in the “init” target. If you call Ant with the arguments “notest j2mepolish”, the test-property will be set to false. This allows controlling the build-script without changing it. In the example the optimization is only done for the device Nokia/6600 when the test-property is true. Also the debug messages are only included when test is true. The obfuscation will only be done when test is false.

You can force a complete rebuild by calling “ant clean”. This can be necessary after you made changes to the device database.

The info-Section

In the info-section general information about the project is defined.

```
<info
    license="GPL"
    name="SimpleMenu"
    version="1.3.4"
    description="A test project"
    vendorName="enough software"
    infoUrl="http://www.enough.de/dictionary"
    icon="dot.png"
    jarName="${polish.vendor}-${polish.name}-example.jar"
    jarUrl="${polish.jarName}"
    copyright="Copyright 2004 enough software. All rights reserved."
    deleteConfirm="Do you really want to kill me?"
    />
```

The <info> element supports following attributes:

info-Attribute	*Required*	*Explanation*
license	Yes	The license for this project. Either "GPL" when the source code of the application is published under the GNU General Public License, or the commercial license key, which can be obtained at www.j2mepolish.org.
name	Yes	The name of the project
version	Yes	The version of the project in the format [major].[minor].[build]. Example: `version="2.1.10"`.
description	Yes	The description of the project. A brief explanation about what this application does should be given here.
vendorName	Yes	The name of the vendor of this application.
jarName	Yes	The name of the jar-files which will be created. You can use following variables: `${polish.vendor}`: The vendor of the device, e.g. Samsung or Motorola `${polish.name}`: The name of the device `${polish.identifier}`: Vendor and device name, e.g. "Nokia/6600" `${polish.version}`: The version of the project as defined above. Example: `jarName="Game-${polish.vendor}-${polish.name}.jar"` results into "Game-Nokia-6600.jar" for an application which has been optimized for the Nokia/6600.
jarUrl	Yes	The URL from which the jar file can be downloaded. This is either the HTTP address, or just the name of the jar-file, when it is loaded locally. Apart from the variables available for the jarName-attribute, you can use the name of the jar-file as defined above: `jarUrl="http://www.enough.de/midlets/Game/${polish.vendor}/${polish.name}/${polish.jarName}"`
copyright	No	Copyright notice.
deleteConfirm	No	The text which is presented to the user when he tries to delete the application.
installNotify	No	A HTTP-URL, which should be called after the successful installation of the application.. This can be useful for tracking how many applications are installed, for example. The user can prevent the install-notify, though. One can use the same variables as for the jarUrl-attribute.
deleteNotify	No	A HTTP-URL, which should be called after the application has been deleted from the device. See the explanation of installNotify.
dataSize	No	The minimum space which is needed on the device, e.g. `dataSize="120kb"`.

The deviceRequirements-Section

The optional <deviceRequirements>-section is responsible for selecting the devices which are supported by the application. When this section is omitted, the application will be optimized for all known devices. Any device capabilities or features can be used for the device selection:

```
<deviceRequirements if="test">
      <requirement name="Identifier" value="Nokia/6600" />
</deviceRequirements>
<deviceRequirements unless="test">
      <requirement name="JavaPackage" value="nokia-ui" />
      <requirement name="BitsPerPixel" value="4+" />
</deviceRequirements>
```

In this example two alternative device-selections are defined – when the test-property is set to true (by defining it with `<property name="test" value="true" />`), only the upper <deviceRequirements>-element is used and the second <deviceRequirements>-element is ignored. The actual requirements are defined with the sub-elements <requirement>. Without any clarification, all listed requirements need to be fulfilled by the device to be selected. There are <or>, <and>, <not> and <xor> elements, which can be used to define the requirements very flexible.

deviceRequirements -Attribute	*Required*	*Explanation*
if	No	The name of the Ant-property which needs to be “true” or “yes” to use this <deviceRequirements>.
unless	No	The name of the Ant-property which needs to be “false” or “no” to use this <deviceRequirements>.

deviceRequirements -Element	*Required*	*Explanation*
requirement	Yes	The requirement which needs to be fulfilled by the device
and	No	Series of requirements, of which all need to be fulfilled.
or	No	Series of requirements, of which at least one needs to be fulfilled.
xor	No	Series of requirements, of which one needs to be fulfilled.
not	No	Series of requirements, of which none must be fulfilled.

The actual work is done by the <requirement> element:

requirement-Attribute	*Required*	*Explanation*
name	Yes	The name of the needed capability, e.g. “BitsPerPixel”.
value	Yes	The needed value of the capability, e.g. “4+” for a color depth or at least 4 bits per pixel.

requirement-Attribute	*Required*	*Explanation*
type	No	The class which controls this requirement. Either a class which extends the `de.enough.polish.ant.requirements.Requirement` class, or one of the base types “Size”, “Int”, “String”, “Version” or “Memory”. Example: `<requirement name="MaxJarSize" value="100+ kb" type="Memory" />`

The <or>, <and>, <not> and <xor> elements can be nested in any manner:

```
<deviceRequirements>
      <requirement name="BitsPerPixel" value="4+" />
      <or>
            <requirement name="JavaPackage" value="nokia-ui, mmapi" />
            <and>
                  <requirement name="JavaPackage" value="mmapi" />
                  <requirement name="JavaPlatform" value="MIDP/2.0+" />
            </and>
      </or>
</deviceRequirements>
```

In this example each supported device must have a color depth of at least 4 bits per pixel. Additionally the device needs to support either the Nokia-UI-API and the Mobile Media-API (mmapi), or the Mobile Media-API and the MIDP/2.0 platform.

Following capabilities can be checked directly:

requirement-name	*Explanation*
BitsPerPixel	Needed color depth of the device: 1 is monochrome, 4 are 16 colors, 8 = 256 colors, 16 = 65.536 colors, 24 = 16.777.216 colors. Example: “4+” for at least 4 bits per pixel or “16” for precisely 16 bits per pixel. `<requirement name="BitsPerPixel" value="4+" />`
ScreenSize	Required width and height of the display, e.g. “120+ x 100+” for a resolution of at least 120 pixels horizontally and 100 pixels vertically. `<requirement name="ScreenSize" value="120+ x 100+" />`
ScreenWidth	The needed horizontal resolution of the display , e.g. “120+” for at least 120 pixels. `<requirement name="ScreenWidth" value="120+" />`
ScreenHeight	The needed vertical resolution of the display, e.g. “100+” for at least 100 pixels. `<requirement name="ScreenHeight" value="100+" />`

requirement-name	*Explanation*
CanvasSize	Required width and height of the MIDP-Canvas. Some devices do not allow the usage of the complete screen. `<requirement name="CanvasSize" value="120+ x 100+" />`
JavaPlatform	The needed platform, e.g. “MIDP/1.0” or “MIDP/2.0+”. `<requirement name="JavaPlatform" value="MIDP/2.0+" />`
JavaPackage	Needed APIs, e.g. “nokia-ui, mmapi”: `<requirement name="JavaPackage" value="nokia-ui, mmapi" />`
JavaProtocol	Needed data exchange protocols, e.g. “serial, socket”: `<requirement name="JavaProtocol" value="serial,socket" />`
HeapSize	The needed heap size of the device, e.g. “200+ kb” or “1.1+ MB” `<requirement name="HeapSize" value="200+kb" />`
Vendor	The vendor of the device, e.g. “Nokia” or “Siemens”. `<requirement name="Vendor" value="Nokia, SonyEricsson" />`
Identifier	The identifier of the device, e.g. “Nokia/6600”. `<requirement name="Identifier" value="Nokia/6600, SonyEricsson/P900" />`
Feature	A feature which needs to be supported by the device. `<requirement name="Feature" value="supportsPointer" />`

The build-Section

With the <build>-section the actual build process is controlled:

```
<build
      symbols="showSplash, AnotherExampleSymbol"
      imageLoadStrategy="background"
      fullscreen="menu"
      usePolishGui="true"
      preverify="/home/user/WTK2.1/bin/preverify"
      >
      <!-- midlets definition -->
      <midlet class="MenuMidlet" name="Example" />
      <!-- project-wide variables - used for preprocessing  -->
      <variables>
            <variable name="update-url"
                  value="http://www.enough.de/update" />
      </variables>
      <!-- obfuscator settings: do not obfuscate when test is true -->
      <obfuscator unless="test" enable="true" name="ProGuard" />
      <!-- debug settings: only debug when test is true -->
      <debug if="test"
             enable="true" visual="false" verbose="true" level="error">
            <filter pattern="de.enough.polish.dict.*" level="debug" />
            <filter pattern="de.enough.polish.ui.*" level="warn" />
      </debug>
</build>
```

Attributes of the build-Section

The build-section has following attributes:

build-Attribute	*Required*	*Explanation*
preverify	Yes	The path to the preverify executable of the Wireless Toolkit. The program is usually within the “bin” folder of the Wireless Toolkit.
sourceDir	No	The path to the source directory. The default path is either “source/src”, “source” or “src”. You can define several paths by separating them with a colon ':' or a semicolon ';': [path1]:[path2]
polishDir	No	The directory containing the sources of J2ME Polish. Defaults to the enough-j2mepolish-build.jar.
symbols	No	Project specific symbols, e.g. “showSplash” which can be checked with “//#ifdef showSplash” in the source code. Several symbols need to be separated by comma.
usePolishGui	No	Defines whether the J2ME Polish GUI should be used at all. Possible values are “true” or “false”. Even if the GUI should be used, it will be used only for devices which have the necessary capabilities (e.g. a color depth of at least 8 bits). Default value is “true”.
fullscreen	No	Defines whether the complete screen should be used for devices which support a full screen mode. Currently these include only devices which support the Nokia-UI API. Possible values are either “no”, “yes” and “menu”. With “yes” the complete screen is used but no commands are supported. With “menu” commands can be used as well. Default setting is “no”.
imageLoadStrategy	No	The strategy for loading pictures. Possible values are either “foreground” or “background”. The “foreground” strategy loads images directly when they are requested. The “background” strategy loads the images in a background-thread. With the background strategy the felt performance of an application can be increased, but not all pictures might be shown right away when the user enters a screen. The definition of the imageLoadStrategy makes only sense when the J2ME Polish GUI is used (usePolishGui=”true”). Default strategy is “foreground”. When the “foreground” strategy is used, the preprocessing symbol “`polish.images.directLoad`” will be defined. When using the “background” strategy, the symbol “`polish.images.backgroundLoad`” is defined.
devices	No	Path to the devices.xml-file. Defaults to devices.xml in the current folder or in enough-j2mepolish-build.jar.

build-Attribute	*Required*	*Explanation*
groups	No	Path to the groups.xml-file. Defaults to groups.xml in the current folder or in enough-j2mepolish-build.jar.
vendors	No	Path to the vendors.xml-file. Defaults to vendors.xml in the current folder or in enough-j2mepolish-build.jar.
apis	No	Path to the apis.xml-file. Defaults to apis.xml in the current folder or in enough-j2mepolish-build.jar.
midp1Path	No	Path to the MIDP/1.0 API. Defaults to "import/midp1.jar".
midp2Path	No	Path to the MIDP/2.0 API. Defaults to "import/midp2.jar".
workDir	No	The temporary build folder. Defaults to "build".
destDir	No	The folder into which the "ready-to-deploy" application bundles should be stored. Defaults to "dist".
apiDir	No	The folder in which the APIs are stored, defaults to "import".
resDir	No	The folder which contains all design definitions and other resources like images, movies etc. By setting a different folder, completely different designs can be demonstrated. Default folder is "resources".
obfuscate	No	Either "true" or "false". Defines whether the applications should be obfuscated. Defaults to "false". Alternatively the nested element "obfuscator" can be used (see below).
obfuscator	No	The name of the obfuscator. Defaults to "ProGuard". Alternatively the nested element "obfuscator" can be used (see below).

Elements of the build-Section

The build section supports several nested elements: <midlet>, <midlets>, <obfuscator>, <variables> and <debug>.

<midlet> and <midlets>

The <midlet>-element defines the actual MIDlet class:

```
<midlet class="de.enough.polish.example.ExampleMidlet" />
```

midlet-Attribute	*Required*	*Explanation*
class	Yes	The complete package and class-name of the MIDlet.
name	No	The name of the MIDlet. Default is the class-name without the package: The MIDlet "com.company.SomeMidlet" is named "SomeMidlet" by default.

midlet-Attribute	*Required*	*Explanation*
icon	No	The icon of this MIDlet. When none is defined, the icon defined in the <info>-section will be used.
number	No	The number of this MIDlet. This is interesting only for MIDlet-suites in which several MIDlets are contained.

The optional <midlets>-element is used as a container for several <midlet>-elements:

```
<midlets>
      <midlet class="de.enough.polish.example.ExampleMidlet" />
      <midlet name="J2ME Polish Demo" number="2" icon="no2.png"
              class="de.enough.polish.example.GuiDemoMidlet"/>
</midlets>
```

<obfuscator>

The optional <obfuscator>-element controls the obfuscating of the application bundle, which decreases the jar-size and makes it difficult to reverse engineer the application.

```
<obfuscator unless="test" enable="true" name="ProGuard" />
```

obfuscator-Attribute	*Required*	*Explanation*
enable	No	Either “true” or “false”. Defaults to “false”. Obfuscating will only be activated when enable=”true”.
if	No	The name of the Ant-property, which needs to be “true” or “yes”, when this <obfuscator> element should be used.
unless	No	The name of the Ant-property, which needs to be “false” or “no”, when this <obfuscator> element should be used.
name	No	The name of the obfuscator which should be used. Defaults to “ProGuard”. Please consider the license agreement of the obfuscator. The ProGuard obfuscator is licensed under the GNU General Public License. The RetroGuard obfuscator is licensed under the GNU Lesser General Public License.
class	No	The class which controls the obfuscator. Each class which extends `de.enough.polish.obfuscate.Obfuscator` can be used. With this mechanism third party obfuscators can be integrated easily.

The <obfuscator> supports the subelement <keep>, which is used to define classes which are loaded dynamically (e.g. with Class.forName(...)) and should not be obfuscated:

```
<obfuscator unless="test" enable="true" name="ProGuard" >
      <keep class="com.company.dynamic.SomeDynamicClass" />
      <keep class="com.company.dynamic.AnotherDynamicClass" />
</obfuscator>
```

keep-Attribute	*Required*	*Explanation*
class	Yes	The full name of the class which should not get obfuscated.

The used MIDlet classes do not need to be set with the <keep> element, since they are saved from obfuscation automatically.

<variables>

The optional <variables>-element contains several variable definitions, which can be used for the preprocessing. This mechanism can be used for example to define configuration values:

```
<variables includeAntProperties="true">
      <variable name="update-url" value="http://www.enough.de/update" />
</variables>
```

variables-Attribute	*Required*	*Explanation*
includeAntProperties	No	Either “true” or “false”. When “true” all Ant-properties will be included and can be used in the preprocessing. Defaults to “false”.

The <variables>-element contains an arbitrary number of <variable>-elements, which define the actual variables. Each variable has the attributes name and value:

variable-Attribute	*Required*	*Explanation*
name	Yes	The name of the variable.
value	Yes	The value of the variable.

Variables which have been defined in this way can be included into the source code with the “//#=” preprocessing directive:

```
//#ifdef update-url:defined
      //#= String url = "${ update-url }";
//#else
      String url = "http://www.default.com/update";
//#endif
```

<debug>

The optional <debug>-element controls the inclusion of debugging messages for specific classes or packages. The debugging messages will be activated or deactivated in the source code, so the performance will not be decreased, when the debugging is deactivated.

```
<debug enable="true" useGui="true" verbose="false" level="error">
      <filter pattern="com.company.package.*" level="info" />
      <filter pattern="com.company.package.MyClass" level="debug" />
</debug>
```

In the source code any debug messages must be accompanied by a //#debug-directive:

```
//#debug
System.out.println("initialization done.");
```

or

```
//#debug warn
System.out.println("could not load something...");
```

In the chapter “The world of preprocessing” on page 30 you will find more about the debugging possibilities.

debug-Attribute	*Required*	*Explanation*
enable	No	Either “true” or “false”. Debugging messages will only be included, when “true” is given.
level	No	The general debug level which is needed for debug messages. Possible values are “debug”, “info”, “warn”, “error”, “fatal” or a user-defined level. Default level is “debug”, so all debugging messages will be included.
verbose	No	Either “true” or “false”. When “true” the time, class-name and line-number will be included in each debugging message. Defaults to “false”. When the verbose mode is enabled, the preprocessing symbol “`polish.debugVerbose`” will be defined. In the verbose mode exceptions thrown by J2ME Polish will contain useful information. Also the key-handling and animation-handling will be monitored and error messages will be given out.
useGui	No	Either “true” or “false”. When “true” all debugging messages, which are passed to the tool `de.enough.polish.util.Debug` will be made available in a form. This can be used to show debugging messages on a real device. Defaults to “false”. When the visual mode is activated, the preprocessing symbol “`polish.useDebugGui`” will be defined.
if	No	The name of the Ant-property, which needs to be “true” or “yes”, when this <debug> element should be used.
unless	No	The name of the Ant-property, which needs to be “false” or “no”, when this <debug> element should be used.

For a finer control of the debugging process, the <debug> element allows the sub-element <filter>, which defines the debug-level for specific classes or packages.

filter-Attribute	*Required*	*Explanation*
pattern	Yes	The name of the class or of the package. When the pattern ends with a star, all classes of that package will be included, e.g. “`com.company.mypackage.*`”

filter-Attribute	*Required*	*Explanation*
level	Yes	The debugging level for all classes with the specified pattern. Possible values are “debug”, “info”, “warn”, “error”, “fatal” or a user-defined level.

The World of Preprocessing

Preprocessing changes the source code before it is compiled. With this mechanism any device optimization can be done easily. An example is the inclusion of the J2ME Polish GUI, which can be included without any intervention of the developer.

Most preprocessing is triggered by directives, which use either symbols or variables. A symbol is like a boolean value – either it is defined or the symbol is not defined. A variable, however, always contains a value, which can be used or compared during the preprocessing.

Symbols and variables are defined in several files:

- build.xml: with the “symbols”-attribute of the <build>-element and with the <variables>-element
- vendors.xml, groups.xml, devices.xml: symbols are defined by the <features>-element, whereas variables are defined by the capabilities. Most capabilities also define symbols, see the section “Capabilities” on page 10.

Preprocessing directives always start with a “//#” and often have one or more arguments. All directives must not span several rows.

J2ME Polish supports all directives of the antenna preprocessor (antenna.sourceforge.net), by the way. So if you migrate from antenna, you can keep your preprocessing directives.

Checking a Single Symbol with #ifdef, #ifndef, #else, #elifdef, #elifndef and #endif

Single symbols can be checked easily with the #ifdef directive:

```
//#ifdef polish.images.directLoad
      Image image = Image.createImage( name );
      //# return image;
//#else
      scheduleImage( name );
      return null;
//#endif
```

When the symbol “polish.images.directLoad” is defined, the code will be transformed to the following:

```
//#ifdef polish.images.directLoad
      Image image = Image.createImage( name );
      return image;
//#else
      //# scheduleImage( name );
      //# return null;
//#endif
```

If, however, the symbol “polish.images.directLoad” is not defined, the transformation will be:

```
//#ifdef polish.images.directLoad
      //# Image image = Image.createImage( name );
      //# return image;
//#else
      scheduleImage( name );
      return null;
//#endif
```

Each #ifdef and #ifndef directive needs to be closed by the #endif directive.
If a variable is defined, it can be checked via `//#ifdef [variable]:defined`:

```
//#ifdef polish.ScreenSize:defined
```

Directive	*Meaning*	*Explanation*
//#ifdef [symbol]	if [symbol] is defined	The symbol named [symbol] needs to be defined, when the next section should be compiled.
//#ifndef [symbol]	if [symbol] is not defined	The symbol named [symbol] must not be defined, when the next section should be compiled.
//#else	else	When the previous section was false, the following section will be compiled (and the other way round).
//#elifdef [symbol]	else if [symbol] is defined	The symbol named [symbol] needs to be defined and the previous section needs to be false, when the next section should be compiled.
//#elifndef [symbol]	else if [symbol] is not defined	The symbol named [symbol] must not be defined and the previous section needs to be false, when the next section should be compiled.
//#endif	end of the if-block	End of every ifdef and ifndef block.

The #ifdef directives can be nested, of course. Other preprocessing directives can be included into the sub-sections as well:

```
//#ifdef mySymbol
      //#ifndef myOtherSymbol
            //#debug
            System.out.println("only mySymbol is defined.");
            doSomething();
      //#else
            //#debug
            System.out.println("mySymbol and myOtherSymbol are defined.");
            doSomethingElse();
      //#endif
//#endif
```

The #ifdef directive and its related directives are faster to process than the more complex #if directives.

Check Several Symbols and Variables with #if, #else, #elif und #endif

With each #ifdef directive only a single symbol can be checked. With the #if and #elif directives, however, complex terms containing several symbols and variables can be evaluated:

```
//#if useEnhancedInput && (polish.pointerSupported || polish.mouseSupported)
      doSomething();
//#endif
```

#if directives can also be nested and contain other preprocessing directives like the #ifdef directives.

#if and #ifdef directives can also be mixed:

```
//#if !basicInput && (polish.pointerSupported || polish.mouseSupported)
      doSomething();
      //#if polish.BitsPerPixel >= 8
            doSomethingColorful();
      //#else
            doSomethingDull();
      //#endif
//#elifdef doWildStuff
      doWildStuff();
//#endif
```

Directive	***Meaning***	***Explanation***
//#if [term]	if [term] is true	The specified term must be true, when the next section should be compiled.
//#else	else	When the previous section was false, the following section will be compiled (and the other way round).
//#elif [term]	else if [term] is true	The specified term needs to be true and the previous section needs to be false, when the next section should be compiled.
//#endif	end of the if-block	End of every if block.

In the terms the boolean operators &&, ||, ^ and ! can be used. Complex terms can be separated using normal parenthesizes "(" and ")". The term arguments for boolean operators are symbols, which are true when they are defined and false otherwise.

Variables can be checked with the comparator ==, >, <, <= and >=. Arguments for the comparators are variables or constants.

A term can include comparators and boolean operators, when the sections are separated by parenthesizes.

Boolean Operator	***Meaning***	***Explanation***

&&	and	Both arguments/symbols need to be defined:
		`true && true = true`
		`true && false = false && true = false && false = false`

||	or	At least one argument/symbol must be defined:
		`true || true = true || false = false || true = true`
		`false || false = false`

^	xor	Only and at least one argument/symbol must be defined:
		`true ^ false = false ^ true = true`
		`true ^ true = false ^ false = false`

Boolean Operator	*Meaning*	*Explanation*

!	not	The argument/symbol must not be defined:
		`! false = true`
		`! true = false`

Comparator	*Meaning*	*Explanation*

==	equals	The left and the right argument must be equal, integers and strings can be compared:
		`8 == 8 = true`
		`Nokia == Nokia = true`
		`//#if polish.BitsPerPixel == 8`
		`//#if polish.vendor == Nokia`

>	greater	The left argument must be greater than the right one. Only integers can be compared:
		`8 > 8 = false`
		`16 > 8 = true`
		`//#if polish.BitsPerPixel > 8`

<	smaller	The left argument must be smaller than the right one. Only integers can be compared:
		`8 < 8 = false`
		`8 < 16 = true`
		`//#if polish.BitsPerPixel < 8`

>=	greater or equals	The left argument must be greater than - or equals - the right one. Only integers can be compared:
		`8 >= 8 = true`
		`16 >= 8 = true`
		`//#if polish.BitsPerPixel >= 8`

<=	smaller or equals	The left argument must be smaller than - or equals - the right one. Only integers can be compared:
		`8 <= 8 = true`
		`8 <= 16 = false`
		`//#if polish.BitsPerPixel <= 8`

Hiding Code

Code sections can be temporarily commented out, to avoid problems in the IDE. A typical problem are several return statements:

```
//#ifdef polish.images.directLoad
      Image image = Image.createImage( name );
      //# return image;
//#else
      scheduleImage( name );
      return null;
//#endif
```

In this example the first return statement is hidden with a “//# “ directive. When the first #ifdef

directive is true, the corresponding code will be made visible again. The space after the # sign is important for the correct handling of such comments.

Debugging with #debug, #mdebug and #enddebug

To include debugging information in a J2ME application is not without problems. The main problem is that any debugging slows down an application unnecessarily. Another problem is that nobody wants the debugging information in an application which should be deployed "in the wild". With J2ME Polish debugging statements can be included into the applications without having the above disadvantages.

The #debug directive is used to include a single line of debugging information:

```
//#debug
System.out.println("debugging is enabled for this class.");
```

You can define what debugging level is used by adding the debugging-level to the #debug directive. When no level is specified the "debug" level is assumed.

```
//#debug info
System.out.println("the info debugging level is enabled for this class.");
```

The #mdebug (multi-line debug) directive can be used to use multiple lines of debugging information. It must be finished with the #enddebug directive:

```
//#mdebug error
e.printStackTrace();
System.out.println("unable to load something: " + e.getMessage() );
//#enddebug
```

Directive	*Explanation*
//#debug [level]	The next line is only compiled when the debugging setting for the class in which the debugging information is enabled for the specified level. When no level is specified, the "debug" level is assumed.
//#mdebug [level]	The next lines up to the #enddebug directive will be compiled only when the debugging setting for the class in which the debugging information is enabled for the specified level. When no level is specified, the "debug" level is assumed.
//#enddebug	Marks the end of the #mdebug section.

Debug Levels

Following debug levels are predefined:

"debug", "info", "warn", "error" and "fatal". The specific level for a class can be defined with the <debug> element of the J2ME Polish-task:

```
<debug enable="true" useGui="true" verbose="false" level="error">
      <filter pattern="com.company.package.*" level="info" />
      <filter pattern="com.company.package.MyClass" level="debug" />
```

```
</debug>
```

Please see the section <debug> in the chapter “The build process” for further information. The levels are weighted, meaning that the debug level is lower than the info level, which is in turn lower than the error level and so forth:

debug < info < warn < error < fatal < user-defined

Thus when the info level is activated for a class, all debugging information with the level “warn”, “error”, “fatal” and with a user-defined level will also be compiled.

User specific debugging levels can be useful for accomplishing specific tasks. For example a level “benchmark” could be defined to allow the measurement of the performance:

```
//#debug benchmark
long startTime = System.currentTimeMilis();
callComplexMethod();
//#debug benchmark
System.out.println("complex method took [" + (System.currentTimeMilis() -
startTime) + "] ms.");
```

The Debug Utility Class

The utility class de.enough.polish.util.Debug can be used for debugging. It is especially useful for the GUI debugging mode, which can be enabled in the build.xml by setting the “useGui” attribute of the <debug>-element to true:

```
<debug enable="true" useGui="true" verbose="false" level="error">
      <filter pattern="com.company.package.*" level="info" />
      <filter pattern="com.company.package.MyClass" level="debug" />
</debug>
```

Its debug-methods are used to either print out the debug messages to the standard output, or to store the messages in a form, which can be shown on a real device. This is especially useful for tracking errors on a real handset:

```
import de.enough.polish.util.Debug;
[...]
try {
      callComplexMethod();
      //#debug info
      Debug.debug( "complex method succeeded." );
} catch (MyException e) {
      //#debug error
      Debug.debug( "complex method failed", e );
}
```

Both debug-methods are used in the example. The second method debug(String, Throwable) also prints out the stack trace of the exception, when the GUI debugging mode is not enabled.

You can make the debug information available in your MIDlet class with the following code:

```
import de.enough.polish.util.Debug;
public class MyMIDlet extends MIDlet {
//#ifdef polish.useDebugGui
private Command logCmd = new Command( "show log", Command.SCREEN, 10 );
//#endif
private Screen mainScreen;
[...]
public MyMIDlet() {
```

```
        [...]
        //#ifdef polish.useDebugGui
        this.mainScreen.addCommand( this.logCmd );
        //#endif
}
public void commandAction(Command cmd, Displayable screen ) {
        [...]
        //#ifdef polish.useDebugGui
        if (cmd == logCmd) {
                this.display.setCurrent( Debug.getLogForm(true, this ) );
        } else if (cmd == Debug.RETURN_COMMAND) {
                this.display.setCurrent( this.mainScreen );
        }
        //#endif
}
```

In the above example the MIDlet “MyMIDlet” adds a command to its main screen when the GUI debugging mode is enabled. When this is the case, the preprocessing symbol “polish.useDebugGui” will be defined. The user can then select the “show log” command to see the form with all logging messages. When the user selects the “return” command from the log-screen, he will return to the main screen.

Please make sure, that you have added the “import/enough-j2mepolish-util.jar” to the classpath of your project, when using the Debug-class.

Debug-Method / Debug-Field	*Explanation*
Debug.debug(String message)	When the GUI debugging mode is activated, the given message-string will be added to the list of messages. Otherwise the message will be printed to the standard output via System.out.println(String).
Debug.debug(String message, Throwable exception)	When the GUI debugging mode is activated, the given message-string will be added to the list of messages. Otherwise the message and the stacktrace of the exception will be printed to the standard output.
Debug.getLogForm(boolean reverse, CommandListener listener)	Retrieves the Form which contains all log messages. When reverse is true, the last log message will be on the top of the form. This method is only available, when the GUI debugging mode is enabled.
Debug.RETURN_COMMAND	The command which is always added the the logging form. This field is only available, when the GUI debugging mode is enabled.

Using Variables with #=

You can add the contents of variables with the #= directive:

```
//#= private String url = ”${ start-url }”;
```

When the variable is not defined, the above example would throw an exception during the preprocessing step of the build process. You can use the “[variable]:defined” symbol, which is set for every known variable:

```
//#ifdef start-url:defined
      //#= private String url = "${ start-url }";
//#else
      private String url = "http://192.168.101.101";
//#endif
```

This is especially useful for setting configuration values, which can change easily, like Web-URLs and so on.

The name of the variable needs to be surrounded by a ${}, just like referring to Ant-properties in the build.xml.

Variables can be defined in the <variables>-element in the build.xml. Other variables are defined in the devices.xml, vendors.xml and groups.xml by the capabilities of the devices, vendors and groups.

Setting CSS Styles with #style

CSS styles are used for the design of the application. The styles are defined in the file polish.css in the resources folder of the project. The developer can control which styles are used for specific Items by using the #style directive:

```
//#style cool, frosty, default
StringItem url = new StringItem( null, "http://192.168.101.101" );
```

In the above example one of the styles “cool”, “frosty” or “default” is used for the new item. The style “frosty” is only used when the style “cool” is not defined. The style “default” is only used, when neither the style “cool” nor the style “frosty” is defined. The style “default” is special, since it is always defined, even when the designer does not define it explicitly. The #style directive needs at least one style name as argument. The styles mentioned here can be defined in the “resources/polish.css” file. In that file the style-names need to start with a dot. In the above example you can define the styles “.cool”, “.frosty” or “default”. The “default” style is a predefined style and its name must not, therefore, start with a dot.

Styles are only used, when the J2ME Polish GUI is used. This can be triggered with the “usePolishGui” attribute of the <build>-element in the build.xml.

Using #style directive improves the performance of the application, since dynamic styles need some processing time. Dynamic styles design items by their position in screens, see section “Dynamic Styles” on page 41.

Styles can be set for all items-constructors and for many methods:

Insertion Point	*Example*	*Explanation*

Item constructors	`//#style cool, frosty, default`
	`StringItem url = new StringItem`
	`( null, "http://192.168.101.101" );`
	`//#style cool`
	`ImageItem img = new ImageItem`
	`( null, iconImage,`
	`ImageItem.LAYOUT_DEFAULT, null );`
		The #style directive can be placed before any item constructor.

Insertion Point	Example	Explanation
Item.setAppearanceMode()	`//#style openLink` `url.setAppearanceMode ( Item.HYPERLINK );`	The #style directive can be placed before calling the setAppearanceMode-method of an Item. Please note, that this method is only available in J2ME Polish.
List.append()	`//#style choice` `list.append( "Start", null  );`	The #style directive can be placed before adding a list element.
List.insert()	`//#style choice` `list.insert( 2, "Start", null  );`	The #style directive can be placed before inserting a list element.
List.set()	`//#style choice` `list.set( 2, "Start", null  );`	The #style directive can be placed before setting a list element.
ChoiceGroup.append()	`//#style choice` `group.append( "Choice 1", null  );`	The #style directive can be placed before adding an element to a ChoiceGroup.
ChoiceGroup.insert()	`//#style choice` `group.insert( 2, "Choice 3", null  );`	The #style directive can be placed before inserting an element to a ChoiceGroup.
ChoiceGroup.set()	`//#style choice` `group.set( 2, "Choice 3", null  );`	The #style directive can be placed before setting an element of a ChoiceGroup.

Exclude or Include Complete Files with #condition

The #condition directive can be used to prevent the usage of complete files, when a condition is not met:

```
//#condition polish.usePolishGui
```

When in the above example the J2ME Polish GUI is not used (and thus the symbol polish.usePolishGui is not defined), the file will not be included into the application bundle for the current device. Conditions can use the same operators and comparators like the #if directive.

Defining Temporary Symbols with #define and #undefine

You can temporarily define and undefine symbols as well. This can be used to evaluate a complex if-term only once and then referring a simple symbol instead of using the complex term again:

```
//#if !basicInput && (polish.pointerSupported || polish.mouseSupported)
      //#define tmp.complexInput
//#endif
```

You can later just check the temporary defined symbol:

```
//#ifdef tmp.complexInput
      doSomethingComplex();
//#endif
```

Directive	*Explanation*
//#define [symbol]	Defines the specified symbol. You can never define the symbol "false".
//#undefine [symbol]	Removes the specified symbol from the pool of defined symbols. You can never undefine the symbol "true".

Please note that you should rely on a defined or undefined symbol only in the same source-file where you defined or undefined that symbol. It is advised that you start the names of defined symbols with "tmp.", so that you always know that this is a temporary symbol.

Inserting Code Fragments with #include

You can insert complete code fragments with the #include directive:

```
//#include ${polish.source}/includes/myinclude.java
```

Within the file-name you can use all defined variables. A useful variable is especially "polish.source", which points the base directory of the file, which is currently preprocessed. I you use only relative names, please bear in mind, that the base directory is the root of your project (or to be more precise: the directory which contains the build.xml file).

Useful Symbols

APIs and MIDP-Version

Please refer to the section "Capabilities" in the chapter "The Device Database" for standard preprocessing symbols.

For each API the device supports a corresponding symbol "polish.api.[name]" is defined:

`polish.api.mmapi` when the Mobile Media API is supported, or
`polish.api.nokia-ui` when the device supports the Nokia UI API.

When the device supports the MIDP/2.0 platform, the symbol "`polish.midp2`" is defined, otherwise the symbol "`polish.midp1`".

J2ME Polish Symbols

Depending on the settings in the build.xml different symbols will be defined. Please compare the "The Build Section", page 19.

Useful Variables

Please refer to the section "Capabilities" on page 10 for standard preprocessing variables.

Other variables include:

- `polish.animationInterval` defines the interval in milliseconds for animations. This defaults to 100 ms, but you can change this value within the <variables> element of the <build>-section.
- `polish.classes.fullscreen` defines the name of the fullscreen-class, when the device supports such a class. For devices which support the Nokia UI API this variable contains "com.nokia.mid.ui.FullCanvas". Following example illustrates the usage:
```
public abstract class MyCanvas
//#if polish.useFullScreen && polish.classes.fullscreen:defined
    //#define tmp.fullScreen
    //#= extends ${polish.classes.fullscreen}
//#else
    extends Canvas
//#endif
```
- `polish.Vendor` defines the vendor of the current device, e.g. "Siemens" or "Nokia":
```
//#if polish.Vendor == Nokia
```
- `polish.Identifier` defines the identifier of the current device, e.g. "Nokia/6600".

File Operations

The variable `polish.source` points the current source directory. This is useful for //#include directives.

The J2ME Polish GUI for Designers

J2ME Polish includes an optional Graphical User Interface, which can be designed using the web-standard Cascading Style Sheets (CSS). So every web-designer can now design mobile applications thanks to J2ME Polish! This chapter will explain all details of the design possibilities, no prior knowledge about CSS is required.[1] The GUI is compatible with the javax.microedition.ui-classes, therefore no changes need to be made in the source code of the application. The GUI will be incorporated by the preprocessing mechanism automatically, unless the "usePolishGui"-attribute of the <build> element is set to false in the build.xml file.

A Quick Glance

All design settings and files are stored in the "resources" directory of the project, unless another directory has been specified.[2] The most important file is `polish.css` in that directory. All design definitions can be found here. The design definitions are grouped in "styles". A style can be assigned to any GUI item like a title, a paragraph or an input field. Within a style several attributes and its values are defined:

```
.myStyle {
      font-color: white;
      font-style: bold;
      font-size: large;
      font-face: proportional;
      background-color: black;
}
```

In this example the style called "myStyle" defines some font values and the color of the background. Any style contains a selector as well as a number of attributes and its values:

Fig. 1: Structure of a style

Each attribute-value pair needs to be finished with a semicolon. The style declaration needs to be finished by a closing curved parenthesis. The selector or name of style is case-insensitive, so ".MySTYle" is the same as ".myStyle".

Apart from the `polish.css` file, you can put images and other contents into the resources-folder. Sub-folders are used for styles and content for specific devices and groups. You can put all resources for Nokia devices into the "Nokia" folder and resources for Samsung's E700 into the "Samsung/E700" folder. This is described in more detail in the "Designing Specific Devices and Device-Groups" section.

You can specify styles directly for GUI items with the #style preprocessing directive in the source code. Alternatively you can use the dynamic names of the GUI items, e.g. "p" for text-items, "a" for hyperlinks or "form p" for all text-items which are embedded in a form. The possible combinations

1 Refer to http://www.w3schools.com/css/ for an excellent tutorial of CSS for web pages.

2 You can specify the directory with the "resDir" attribute of the <build> element in the build.xml file. This can be used to create completely different designs for one application.

as well as the predefined style-names are discussed in the section “Dynamic, Static and Predefined Styles”.

Styles can extend other styles with the extends-keyword, e.g. “.myStyle extends baseStyle {}”. This process is described in the section “Extending Styles”.

J2ME Polish supports the CSS box model with margins, paddings and content. Other common design settings include the background, the border and font-settings. These common settings are described in the section “Common Design Attributes”. Attributes for specific GUI items as well as the details of the different background and border types are discussed in the section “Specific Design Attributes”.

Designing for Specific Devices or Device-Groups

Sometimes the design needs to be adapted to a specific device or a group of devices. You can easily use specific pictures, styles etcetera by using the appropriate sub folders of the “resources” folder.

The Hierarchy of the “resources” Folder

In the resources folder itself you put all resources and design definitions which should be valid for all devices.

In the folder named like the vendor of a device (e.g. “Nokia”, “Samsung” or “Motorola”) you put all resources and design definitions for devices of that vendor.

In the folder named like the explicit and implicit groups of a device you add the resources and design definitions for these device-groups. An explicit group is for example the “Series60” group, implicit groups are defined by the supported APIs of a device and the BitsPerPixel capability of devices. You can add a small movie for all devices which support the Mobile Media API (mmapi) by putting that movie into the “resources/mmapi” folder. Or you can add colored images for all devices which have at least a color depth of 8 bits per pixel by putting these images into the “resources/BitsPerPixel8+” folder.

Last but not least you can use device specific resources and design definitions by putting them into the “resources/[vendor]/[device]” folder, e.g. “resources/Nokia/6600” or “resources/Samsung/E700”.

Any existing resources will be overwritten by more specific resources:

1. At first the basic resources and definitions found in the “resources” folder will be used.
2. Secondly the vendor-specific resources will be used, e.g. “resources/Nokia”.
3. Thirdly the group-specific resources will be used, e.g. “resources/mmapi”, “resources/Series60”, “resources/BitsPerPixel.8+” or “resources/BitsPerPixe.16”.
4. The resources and settings in the device specific folder will overwrite all other resources and settings. The device specific folder is for example the folder “resources/Nokia/6600” for the Nokia/6600 phone or the folder “resources/Samsung/E700” for Samsung's E700.

When you add the polish.css file for a specific vendor, group or device, you do not need to repeat all styles and attributes from the more basic settings. You need to specify the more specific setting only. When you want to change the color of a font, you just need to specify the “font-color”

attribute of that style. No other attributes or styles need to be defined. This is the cascading character of the Cascading Style Sheets of J2ME Polish.

This example illustrates the cascading character of polish.css:

In “resources/polish.css” you define the style “myStyle”:

```
.myStyle {
      font-color: white;
      font-style: bold;
      font-size: large;
      font-face: proportional;
      background-color: black;
}
```

You can change the font-color of that style for all Nokia devices with the following declaration in “resources/Nokia/polish.css”:

```
.myStyle {
      font-color: gray;
}
```

You can specify another font-size and font-color for the Nokia 6600 phone with these settings in “resources/Nokia/6600/polish.css”:

```
.myStyle {
      font-color: red;
      font-size: medium;
}
```

Groups

Every device can have explicit and implicit groups. Explicit groups are stated by the <groups> element of the device in the file devices.xml[3]. Implicit groups are defined by the capabilities of the device: Each supported API results in an implicit group and the BitsPerPixel capability results in several groups.

API- and Java-Platform-Groups

A device can support different APIs and Java-platforms.

When the device supports the MIDP/1.0 standard, it belongs to the “midp1”-group, otherwise it belongs to the “midp2”-group. So you can specify the layout of MIDP/1.0 devices in “resources/midp1/polish.css”. And you can use specific images or other resources for MIDP/2.0 devices in the folder “resources/midp2”. The supported platform of a device can be specified in the devices.xml file with the <JavaPlatform> element. Alternatively this setting can be specified in the file groups.xml for specific groups.

For each supported API an implicit group is created. When the device supports the Mobile Media API (mmapi), it belongs to the “mmapi”-group. When the device supports the Nokia-UI API, it belongs to the “nokia-ui” group. The name of the implicit group is defined by the <symbol> element of the API in the file apis.xml.

BitsPerPixel-Groups

Every device display has a specific color depth which is specified by the BitsPerPixel-capability of

3 devices.xml can either be found in the root folder of the project or in the import/enough-j2mepolish-build.jar file.

that device in the devices.xml file. Depending on how many bits per pixel are supported, the device belongs to different groups:

Bits per Pixel	*Colors*	*Groups*
1	$2^1 = 2$ (b/w)	BitsPerPixel.1
4	$2^4 = 16$	BitsPerPixel.4 BitsPerPixel.4+
8	$2^8 = 256$	BitsPerPixel.8 BitsPerPixel.8+ BitsPerPixel.4+
12	$2^{12} = 4.096$	BitsPerPixel.12 BitsPerPixel.12+ BitsPerPixel.8+ BitsPerPixel.4+
16	$2^{16} = 65.536$	BitsPerPixel.16 BitsPerPixel.16+ BitsPerPixel.12+ BitsPerPixel.8+ BitsPerPixel.4+
24	$2^{24} = 16.777.216$	BitsPerPixel.24 BitsPerPixel.24+ BitsPerPixel.16+ BitsPerPixel.12+ BitsPerPixel.8+ BitsPerPixel.4+

So you can put images for phones with at least 16 colors into the “resources/BitsPerPixel.4+” folder. And you can specify settings for true color devices in the file “resources/BitsPerPixel.24/polish.css”.

Dynamic, Static and Predefined Styles

J2ME Polish distinguishes between dynamic, static and predefined styles:

- Predefined styles are used by the GUI for several items like screen-titles.
- Static styles are defined in the source code of the application with the #style preprocessing directive.
- Dynamic styles are used for items according to their position on the screen.

Static Styles

The easiest styles are the static ones. The programmer just needs to tell the designer the style names and what they are used for (that is for what kind of items or screens) and the designer defines them in the appropriate polish.css file. Static styles always start with a dot, e.g. “.myStyle”.

Static styles are faster than dynamic styles. It is therefore recommended to use static styles whenever possible.

Predefined Styles

Predefined styles are static styles which are used by the J2ME Polish GUI. In contrast to the normal "user-defined" static styles their names do not start with a dot, e.g. "title" instead of ".title".

Following predefined styles are used:

Style	*Description*
title	The style of screen-titles.
focused	The style of a currently focused item. This style is used in Lists, Forms and for Containers like the ChoiceGroup.
menu	This style is used for designing the menu bar in the full screen mode. The full screen mode can be triggered by the "fullScreenMode" attribute of the <build> element in the build.xml (with fullScreenMode="menu"). In the menu style you can also define which style is used for the currently focused command with the "focused-style"-attribute, e.g. "focused-style: menuFocused;". In this case you need to define the static style ".menuFocused" as well.
menuItem	The style used for the menu items (the commands) of a screen. When menuItem is not defined, the "menu" style is used instead.
default	The style which is used by the J2ME Polish GUI when the desired predefined style is not defined. The default style is always defined, even when it is not explicitly defined in the polish.css file.

The names of predefined styles must not be used for static styles, so you must not use a static style with the name ".title" etc.

Dynamic Styles

Dynamic styles can be used to apply styles to items without using #style directives in the source code. With dynamic styles the designer can work completely independent of the programmer and try out new designs for GUI items which have not yet an associated static style. You can also check out the power and possibilities of the J2ME Polish API without changing the source code of an existing application at all.

Obviously, dynamic styles need a bit more memory and processing time than static styles. It is recommended, therefore, to use static styles instead for finished applications.

Dynamic styles do not start with a dot and use the selectors of the items they want to design:

Texts use either "p", "a", "button" or "icon". Screens use the name of the screen, e.g. "form", "list" or "textbox".

```
p {
      font-color: black;
```

```
      font-size: medium;
      background: none;
}
form {
      margin: 5;
      background-color: gray;
      border: none;
      font-size: medium;
}
```

You can also design only items which are contained in other items or screens:

The style “form p” designs all text-items (of the class StringItem) which are contained in a form:

```
form p {
      font-color: white;
      font-size: medium;
}
```

Static styles and dynamic styles can be used together, you can design all hyperlinks[4] in the screen with the style “.startScreen” for example with the following style declaration:

```
.startScreen a {
      font-color: blue;
      font-size: medium;
      font-style: italic;
}
```

Items and screens have specific selectors for dynamic styles:

Item-Class	*Selector*	*Explanation*
StringItem	p	StringItem shows text. The “p” selector is used, when the item has the appearance mode PLAIN.
	a	The “a” selector is used, when the item has the appearance mode HYPERLINK.
	button	The “button” selector is used, when the item has the appearance mode BUTTON.
ImageItem	img	Shows an image.
Gauge	gauge	Shows a progress indicator.
Spacer	spacer	Is used for showing an empty space. The usage of the Spacer item is discouraged, since the spaces can be set for all items with the margin and padding attributes.
IconItem	icon	Shows an image together with text.
TextField	textfield	Allows textual input from the user.
DateField	datefield	Allows the input of dates or times from the user.
ChoiceGroup	choicegroup	Contains several choice items.

4 StringItems which have the appearance mode Item.HYPERLINK

Item-Class	*Selector*	*Explanation*
ChoiceItem	listitem	Shows a single choice. The selector “listitem” is used, when this item is contained in an implicit list.
	radiobox	The selector “radiobox” is used when the list or choice group has the type “exclusive”.
	checkbox	The selector “checkbox” is used when the list or choice group has the type “multiple”.
	popup	The selector “popup” is used when the choice group has the type “popup”.

Screen-Class	*Selector*	*Explanation*
List	list	Shows several choice items.
Form	form	Contains different GUI items.
TextBox	textbox	Contains a single textfield.

Extending Styles

A style can extend another style. It inherits all the attributes of the extended style. With this mechanism a lot of writing work can be saved:

```
.mainScreen {
      margin: 10;
      font-color: black;
      font-size: medium;
      font-style: italic;
      background-color: gray;
}
.highscoreScreen extends mainScreen {
      font-color: white;
      background-color: black;
}
```

In the above example the style “highscoreScreen” inherits all attributes of the “mainScreen” style, but “font-color” and “background-color” are specified differently.

Circle inheritance is not allowed, so the following example results in a build error:

```
.baseScreen extends highscoreScreen {  /* this extends is invalid! */
      margin: 5;
      font-color: white;
}
.mainScreen extends baseScreen {
      margin: 10;
      font-color: black;
      font-size: medium;
      font-style: italic;
      background-color: gray;
}
.highscoreScreen extends mainScreen {
```

```
      font-color: white;
      background-color: black;
}
```

The above example would be valid, when the style “baseScreen” would not extend the “highscoreScreen”-style.

CSS Syntax

Following rules do apply for CSS styles:

Structure of a CSS Declaration

Fig. 2: Structure of a style

Every style starts with the selector followed by an opening curved parenthesis, a number of attribute-value pairs and a closing curved parenthesis.

The selector can consist of several item-names and contain an “extends” clause.

Each attribute-value pair needs to be finished by a semicolon.

Naming

Styles can use any names, as long as they consist of alphanumeric and underline (_) characters only. Names are not case-sensitive. Static styles need to start with a dot. Static styles must not use the names of dynamic or predefined styles. All Java keywords like “class”, “int” or “boolean” etcetera are not allowed as style names.

Grouping of Attributes

Attributes can be grouped for easier handling:

```
.mainScreen {
      font-color: black;
      font-size: medium;
      font-style: italic;
      font-face: system;
}
```

The above code is equivalent with the following:

```
.mainScreen {
      font {
            color: black;
            size: medium;
            style: italic;
            face: system;
      }
}
```

The grouping makes the declarations better readable for humans.

Referring to Other Styles

When another style is referred, the dots of static styles do not need to be written. Styles can be referred in attributes or after the extends keyword in the selector of a style.

Comments

Comments can be inserted at any place and start with "/*" and stop with "*/". Everything between these boundaries is ignored:

```
/* this style designs the main screen: */
.mainScreen {
      /* defining the color of a font: */
      font-color: black;
      /* sizes are small, medium and large: */
      font-size: medium;
      /* styles are plain, bold, italic or underlined: */
      font-style: italic;
      /* the face can either be system, proportional or monospace: */
      font-face: system;
}
```

Common Design Attributes

Structure of polish.css

The polish.css file can contain different sections:

- colors: The colors-section contains the definition of colors.
- fonts: The fonts-section contains font definitions.
- backgrounds: The backgrounds-section contains background definitions.
- borders: The borders-section contains definition of borders.
- rest: The rest of polish.css contains the actual style definitions.

The defined colors, fonts, backgrounds and borders can be referenced in the actual style definitions. This makes changes very easy, since you need to change the value only in one position.

Structure of a Style Definition

Each style can contain different "sections":

- margin: The gap between items
- padding: The gap between the border and the content of an item
- font: The used content-font and its color
- label: The used label-font and its color
- layout: The layout of the items.
- background: The definition of the item's background
- border: The definition of the item's border
- before and after: Elements which should be inserted before or after the items.

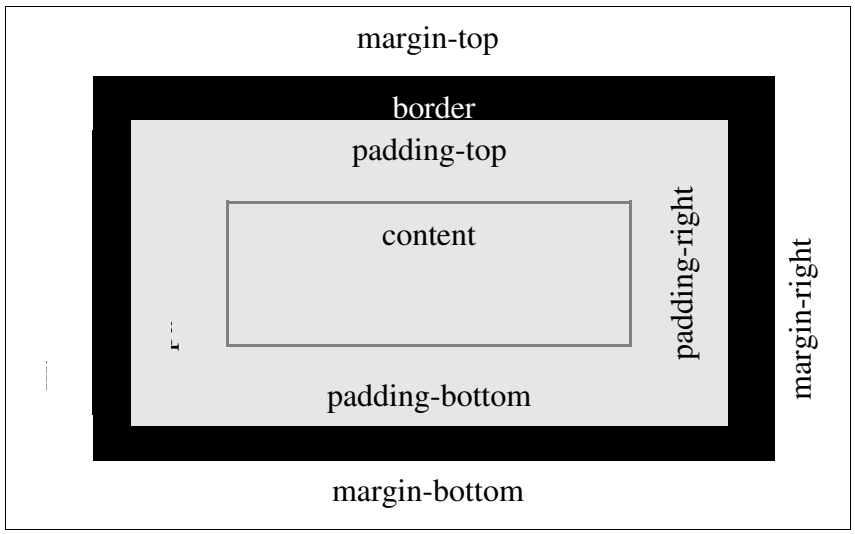
– specific attributes: All attributes for specific GUI items.

An example of such a complete style definition is the following:

```
/* this style designs the currently focused element in a list, form etc: */
focused {
      /* margins and paddings: */
      margin: 2;
      margin-left: 5;
      margin-right: 10;
      padding: 1;
      padding-vertical: 2;
      /* font and label: */
      font {
            color: blue;
            size: medium;
            face: system;
      }
      label {
            color: black;
            size: small;
      }
      /* layout is centered: */
      layout: center;
      /* background: */
      background-color: gray;
      /* no border: /*
      border: none;
      /* before: add an image: */
      before: url( arrow.png );
      /* after: add another image: */
      after: url( leftArrow.png );
      /* no specific attributes are used in this example*/
}
```

The CSS Box Model: Margins and Paddings

All GUI items support the standard CSS box model:



The margin describes the gap to other GUI items. The padding describes the gap between the border

of the item and the actual content of that item. So far no different border-widths (for left, right, top and bottom) can be set with J2ME Polish. Since this is a more bizarre and seldom used feature, not much harm is done.

The margin- and padding-attributes define the default gaps for the left, right, top and bottom elements. Any margin has the default value of 0 pixels, whereas any padding defaults to 1 pixel. Next to the left, right, top and bottom padding, J2ME Polish also knows the vertical and the horizontal paddings. These define the gaps between different content sections. The gap between the label of an item and the actual content is defined by the horizontal padding. Another example is the icon, which consists of an image and a text. Depending on the align of the image, either the vertical or the horizontal padding fills the space between the icon-image and the icon-text.

In the following example, the top, right and bottom margin is 5 pixels, whereas the left margin is 10 pixels:

```
.myStyle {
      margin: 5;
      margin-left: 10;
      font-color: black;
}
```

Percentage values can also be used. Percentage values for top, bottom and vertical attributes relate to the height of the display. Percentage values for left, right and horizontal attributes relate to the width of the display:

```
.myStyle {
      padding-left: 2%;
      padding-right: 2%;
      padding-vertical: 1%;
      font-color: black;
}
```

When the device has a width of 176 pixels, a padding of 2% results into 3.52 pixels, meaning effectively a padding of 3 pixels. At a display height of 208 pixels a vertical padding of 1% results into a padding of 2.08 pixels or effectively 2 pixels. Please note that the capability “ScreenSize” of the device needs to be defined when you use percentage values.

The Layout Attribute

The layout attribute defines how the affected item should be aligned and laid out. Possible layout values are for example left, right or center. All layout values of the MIDP/2.0 standard can be used:

Layout	*Alternative Names*	*Explanation*
left	-	The affected items should be left-aligned.
right	-	The affected items should be right-aligned.
center	horizontal-center, hcenter	The affected items should be centered horizontally.
expand	horizontal-expand, hexpand	The affected items should use the whole available width (i.e. should fill the complete row).

Layout	*Alternative Names*	*Explanation*
shrink	horizontal-shrink, hshrink	The affected items should use the minimum width possible.
top	-	The affected items should be top-aligned.
bottom	-	The affected items should be bottom-aligned.
vcenter	vertical-center	The affected items should be centered vertically.
vexpand	vertical-expand	The affected items should use the whole available height (i.e. should fill the complete column).
vshrink	vertical-shrink	The affected items should use the minimum height possible.
newline-after	-	Items following an item with a newline-after layout should be placed on the next line. Currently the newline settings will be ignored, since every item will be placed on a new line.
newline-before	-	The affected items should always start on a new line (when there are any items in front of it). Currently the newline settings will be ignored, since every item will be placed on a new line.
plain	default, none	No specific layout should be used, instead the default behavior should be used. Such a layout does not need to be defined explicitly, but it can be useful to overwrite a basic setting.

Layout values can also be combined, using either the “||”, “|”, “or“ or “and“ operators. All operators result in the same combination. An item can be centered and using the whole available width with following example:

```
.myStyle {
      layout: center | expand;
}
```

This is equivalent with:

```
.myStyle {
      layout: center || expand;
}
```

And equivalent with:

```
.myStyle {
      layout: center and expand;
}
```

And equivalent with:

```
.myStyle {
      layout: center or expand;
}
```

And equivalent with:

```
.myStyle {
      layout: hcenter | hexpand;
}
```

And equivalent with:

```
.myStyle {
      layout: horizontal-center | horizontal-expand;
}
```

Colors

Colors can be defined in the colors section and in each attribute which ends on "-color", e.g. "font-color", "border-color" etc.

Predefined Colors

The 16 standard windows colors are predefined:

Color	*Hex-Value*	*Example*	*Color*	*Hex-Value*	*Example*
white	#FFFFFF		yellow	#FFFF00	
black	#000000		maroon	#800000	
red	#FF0000		purple	#800080	
lime	#00FF00		fuchsia	#FF00FF	
blue	#0000FF		olive	#808000	
green	#008000		navy	#000080	
silver	#C0C0C0		teal	#008080	
gray	#808080		aqua	#00FFFF	

Another predefined color is "transparent", which results in an transparent area. "transparent" is only supported by some GUI elements like the menu-bar of a full-screen menu.

The colors Section

The colors section of the polish.css file can contain colors, which can be referenced in the styles, fonts, border and background sections. You can even overwrite the predefined colors to confuse other designers!

```
colors {
      bgColor: #50C7C7;
      bgColorLight: #50D9D9;
      gray: #7F7F7F;
}
```

How to Define Colors

A color can be defined in many different ways:

```
.myStyle {
      font-color: white;      /* the name of the color */
      border-color: #80FF80;  /* a rgb hex value */
```

```
      start-color: #F00;      /* a short rgb-hex-value - this is red */
      menu-color: #7F80FF80;  /* an alpha-rgb hex value */
      background-color: rgb( 255, 50, 128 );  /* a rrr,ggg,bbb value */
      fill-color: rgb( 100%, 30%, 50% );      /* a rgb-value with percentage */
      label-color: argb( 128, 255, 50, 128 ); /* a aaa, rrr, ggg, bbb value */
}
```

Color *names* refer to one of the predefined colors or a color which has been defined in the colors-section:
`color: black;` or `color: darkBackgroundColor;`

The *hex-value* defines a color with two hexadecimal digits for each color (RRGGBB). Additionally the alpha blending-component can be added (AARRGGBB).
`color: #FF000;` defines red. color: `#7FFF0000;` defines a half transparent red.

The shortened hex-value defines a color by a RGB-value in hexadecimal. Every digit will be doubled to retrieve the full hex-value:
`color: #F00;` is equivalent with `color: #FF0000;`
`color: #0D2;` is equivalent with `color: #00DD22;` and so on.

A rgb-value starts with "rgb(" and then lists the decimal value of each color from 0 up to 255:
`color: rgb( 255, 0, 0 );` defines red. `color: rbg( 0, 0, 255);`defines blue and so on.

Alternatively percentage values can be used for rgb-colors:
`color: rgb( 100%, 0%, 0% );` defines red as well as `color: rgb( 100.00%, 0.00%, 0.00% );`

Alpha-RGB colors can be defined with the argb()-construct:
`color: argb( 128, 255, 0, 0 );` defines a half transparent red. For the argb-construct percentage values can be used as well.

Alpha Blending

Colors with alpha blending can be defined with hexadecimal or argb-definitions (see above). An alpha value of 0 results in fully transparent pixels, whereas the value FF (or 255 or 100%) results in fully opaque pixels. Some devices support values between 0 and FF, which results in transparent colors. Colors with a specified alpha channel can only be used by specific GUI items. Please refer to the documentation of the specific design attributes.

Fonts and Labels

Many GUI Items have text elements, which can be designed with the font- or label-attributes.

Both attribute-groups support the same attributes:

Attribute	***Possible Values***	***Explanation***
color	Reference to a color or direct declaration of the color.	Depending on the number of colors the device supports, colors can look differently on the actual device.

Attribute	*Possible Values*	*Explanation*
face	system (default, normal)	The default font-face which is used when the font-face or label-face attribute is not set.
	proportional	A proportional face. This is on some devices actually the same font-face as the system-font.
	monospace	A font-face in which each character has the same width.
size	small	The smallest possible font.
	medium (default, normal)	The default size for texts.
	large (big)	The largest possible font size.
style	plain (default, normal)	The default style.
	bold	A bold thick style.
	italic (cursive)	A cursive style.
	underlined	Not really a style, just an underlined text.

An example font and label specification:

```
.myStyle {
      font-color: white;
      font-face: default; /* same as system or normal */
      font-size: default; /* same as medium or normal */
      font-style: bold;
      label-color: gray;
      label-face: proportional;
      label-size: small;
      label-style: cursive; /* same as italic */
}
```

The same specification can also be written as follows:

```
.myStyle {
      font {
            color: white;
            style: bold;
      }
      label {
            color: gray;
            face: proportional;
            size: small;
            style: cursive; /* same as italic */
      }
}
```

In the font-definition the above face and size attributes can be skipped, since they only define the default behavior anyhow.

Backgrounds and Borders

Each style can define a specific background and border. There are many different types of backgrounds and borders available, of which some are even animated.

A specification of a simple background and border is the following example:

```
.myStyle {
      background-color: white;
      border-color: gray;
      border-width: 2;
}
```

This example creates a white rectangular background with a gray border, which is 2 pixel wide.

```
.myStyle {
      background {
            type: pulsating;
            start-color: white;
            end-color: pink;
            steps: 30;
      }
}
```

The above example creates a background which color is constantly changing between white and pink. 30 color shades are used for the animation.

When no background or border should be used, the “none” value can be set:

```
.myStyle {
      background: none;
      border: none;
}
```

If more complex types should be used, the background- or border-type needs to be specified explicitly. The following example illustrates this for an background, which colors change all the time:

The available background- and border-types are explained in detail in the section “Specific Design Attributes”.

Before and After Attributes

The before and after attributes can be used to insert content before or after GUI items which have the specified style.

The following example adds a heart picture after the actual GUI items. The “focused” style is a predefined style which is used for lists, forms, and so on.

```
focused {
```

```
        after: url( heart.png );
        background: none;
        border-type: round-rect;
        border-arc: 6;
        border-color: yellow;
        border-width: 2;
        layout: left | expand;
        font-color: black;
}
```

Fig. 3: The after attribute in action.

Currently only images can be included.

Specific Design Attributes

Many GUI items support specific CSS attributes.

Backgrounds

There are many different background-types which make use of specific CSS attributes. When another background than the default simple background or the image background should be used, the background-type attribute needs to be declared.

When no background at all should be used, the `background: none;` declaration can be used.

Simple Background

The simple background just fills the background with one color. When no background type is specified, the simple background is used by default, unless the “background-image” attribute is set. In the later case the image background will be used.

The simple background supports the color attribute:

Attribute	*Required*	*Explanation*
color	Yes	The color of the background, either the name of the color or a direct definition.

The following styles use a yellow background:

```
.myStyle {
        background-color: yellow;
}
.myOtherStyle {
        background-color: rgb( 255, 255, 0 );
}
```

Round-Rect Background

The round-rect background paints a rectangular background with round edges. It supports following attributes:

Attribute	*Required*	*Explanation*
type	Yes	The type needs to be “round-rect” or “roundrect”.

Attribute	*Required*	*Explanation*
color	No	The color of the background, either the name of the color or a direct definition. The default color is white.
arc	No	The diameter of the arc at the four corners. Defaults to 10 pixels, when none is specified.
arc-width	No	The horizontal diameter of the arc at the four corners. Defaults to the arc-value, when none is specified.
arc-height	No	The vertical diameter of the arc at the four corners. Defaults to the arc-value, when none is specified.

This example creates a purple background with an arc diameter of 6 pixels:

```
.myStyle {
      background {
            type: round-rect;
            color: purple;
            arc: 6;
      }
}
```

The following example uses a different horizontal arc diameter:

```
.myStyle {
      background-type: round-rect;
      background-color: purple;
      background-arc: 6;
      background-arc-width: 10;
}
```

Image Background

The image background uses an image for painting the background. This background type is used by default when no type is set and the “background-image” attribute is set. The background supports following attributes:

Attribute	*Required*	*Explanation*
type	No	When used needs to be “image”.
color	No	The color of the background, either the name of the color, a direct definition or “transparent”. The default color is white. This color can only be seen when the image is not big enough.
image	Yes	The URL of the image, e.g. “url(background.png)”
repeat	No	Either “repeat”, “no-repeat”, “repeat-x” or “repeat-y”. Determines whether the background should be repeated, repeated horizontally or repeated vertically. Default is no repeat.

A background image, which should not be repeated will be centered automatically. The following style uses the image “bg.png” as a background:

```
.myStyle {
      background-image: url( bg.png );
}
```

This style uses the image “heart.png” as a repeated background:

```
.myStyle {
      background-image: url( heart.png );
      background-repeat: repeat;
}
```

Pulsating Background

The pulsating background animates the color of the background. The color is changing from a start-color to an end-color. It supports following attributes:

Attribute	*Required*	*Explanation*
type	Yes	The type needs to be “pulsating”.
start-color	Yes	The color of the background at the beginning of the animation sequence.
end-color	Yes	The color of the background at the end of the animation sequence.
steps	Yes	Defines how many color-shades between the start- and the end-color should be used.
repeat	No	Either “yes”/”true” or “no”/”false”. Determines whether the animation should be repeated. Defaults to “yes”.
back-and-forth	No	Either “yes”/”true” or “no”/”false. Determines whether the animation sequence should be running backwards to the start-color again, after it reaches the end-color. When “no” is selected, the animation will jump from the end-color directly to the start-color (when repeat is enabled). Defaults to “yes”.

The following style starts with a white background and stops with a yellow background:

```
.myStyle {
      background {
            type: pulsating;
            start-color: white;
            end-color: yellow;
            steps: 15;
            repeat: false;
            back-and-forth: false;
      }
}
```

Borders

There are many different border-types which make use of specific CSS attributes. When another border than the default simple border should be used, the border-type attribute needs to be declared.

When no border at all should be used, the `border: none;` declaration can be used.

Simple Border

The simple border paints a rectangle border in one color. The type attribute does not need to be set for the simple border, since this is the default border. The only supported attributes are the color and the width of the border:

Attribute	*Required*	*Explanation*
color	Yes	The color of the border, either the name of the color or a direct definition.
width	No	The width of the border in pixels. Defaults to 1.

The following style uses a black border which is 2 pixels wide:

```
.myStyle {
      border-color: black;
      border-width: 2;
}
```

Round-Rect Border

The round-rect border paints a rectangular border with round edges. It supports following attributes:

Attribute	*Required*	*Explanation*
type	Yes	The type needs to be “round-rect” or “roundrect”.
color	Yes	The color of the border, either the name of the color or a direct definition.
width	No	The width of the border in pixels. Defaults to 1.
arc	No	The diameter of the arc at the four corners. Defaults to 10 pixels, when none is specified.
arc-width	No	The horizontal diameter of the arc at the four corners. Defaults to the arc-value, when none is specified.
arc-height	No	The vertical diameter of the arc at the four corners. Defaults to the arc-value, when none is specified.

This example creates a 2 pixels wide purple border with an arc diameter of 6 pixels:

```
.myStyle {
      border {
            type: round-rect;
            color: purple;
            width: 2;
            arc: 6;
      }
}
```

The following example uses a different horizontal arc diameter:

```
.myStyle {
      border-type: round-rect;
      border-color: purple;
      border-width: 2;
      border-arc: 6;
      border-arc-width: 10;
}
```

Shadow Border

The shadow border paints a shadowy border. Following attributes are supported:

Attribute	*Required*	*Explanation*
type	Yes	The type needs to be “shadow”, “bottom-right-shadow” or “right-bottom-shadow”.
color	Yes	The color of the border, either the name of the color or a direct definition.
width	No	The width of the border in pixels. Defaults to 1.
offset	No	The offset between the corner and the start of the shadow. Defaults to 1 pixel, when none is specified.

Currently only a “bottom-right-shadow” is supported, so the border is painted below the item and right of the item.
The following example uses a green shadow border:

```
.myStyle {
      border-type: shadow;
      border-color: purple;
      border-width: 2;
      border-offset: 2;
}
```

Screens: List, Form and TextBox

Predefined Styles for Lists, Forms and TextBoxes

All screens have a title and one or several embedded GUI items. In a Form or List one item is usually focused. Screens can also have a designable menu, when the application uses a full-screen-mode (on Nokia devices)[5]. Some of the used styles can also use additional attributes.

Style-Name	*Add. Attributes*	*Explanation*
title	-	The title of a screen.
focused	-	The style of the currently focused item in the screen.

5 The fullscreen-mode can be activated by setting the <build> attribute “fullscreen” to either “menu” or “yes” in the build.xml file.

Style-Name	*Add. Attributes*	*Explanation*
menu		The style of the full-screen menu.
	focused-style	The name of the style for the currently focused menu item.
	menubar-color	The background color of the menu-bar. Either the name or the definition of a color or “transparent”. Defaults to “white”.
	label-color, label-face, label-size, label-style	The font of the menu-commands (like “Select” or “Cancel”). Default color is “black”, default font is the system font in a bold style and medium size.
menuItem	-	The style for the commands in the menu. When not defined, the menu-style will be used.

Additional Attributes for Screens

Each screen itself can have some additional attributes:

Attribute	*Required*	*Explanation*
focused-style	No	The name of the style for the currently focused menu item. Defaults to the predefined “focused” style.
columns	No	The number of columns. This can be used to layout the items in a table. Defaults to 1 column.
columns-width	No	Either “normal”, “equal” or the width for each column in a comma separated list (e.g. “`columns-width: 60,60,100;`”). Defaults to “normal”, meaning that each column uses as much space as the widest item of that column needs. The “equal” width leads to columns which have all the same width. The explicit list of column-widths results in the usage of those widths.

The following example uses these additional attributes:

```
list {
      background-image: url( bg.png );
      columns: 2;
      columns-width: equal;
      focused-style: .listFocusStyle; /* this style needs to
be defined, too */
}
```

Fig. 4: 2 columns instead of a normal list.

Dynamic Styles for Screens

Dynamic styles can be used when no styles are explicitly set in the application code (compare page 41).

List

A list uses the dynamic selector “list” and always contains choice-items. The selector of these items depends on the type of the list. An implicit list contains a “listitem”, a multiple list contains a “checkbox” and an exclusive list contains a “radiobox”.

```
list {
      background-color: pink;
}
```

defines the background color for screens of the type “List”.

```
listitem { /* you could also use "list listitem" */
      font-style: bold;
}
```

defines the font style for the items in an implicit list.

Form

A form uses the dynamic selector “form” and can contain different GUI items.

```
form {
      background-color: pink;
}
```

defines the background color for screens of the type “Form”.

```
form p {
      font-style: bold;
}
```

defines the font style for normal text items in a “Form”.

TextBox

A TextBox uses the dynamic selector “textbox” and contains a single “textfield” item.

Example

The following example designs the main menu of an application, which is implemented using a List.

```
colors {
      pink:  rgb(248,39,186);
      darkpink:  rgb(185,26,138);
```

```
}

menu {
	margin-left: 2;
	padding: 2;
	background {
		type: round-rect;
		color: white;
		border-width: 2;
		border-color: yellow;
	}
	focused-style: .menuFocused;
	menubar-color: transparent;
	menufont-color: white;
}

menuItem {
	margin-top: 2;
	padding: 2;
	padding-left: 5;
	font {
		color: black;
		size: medium;
		style: bold;
	}
	layout: left;
}

.menuFocused extends .menuItem {
	background-color: black;
	font-color: white;
	layout: left | horizontal-expand;
	after: url(heart.png);
}

title {
	padding: 2;
	margin-top: 0;
	margin-bottom: 5;
	margin-left: 0;
	margin-right: 0;
	font-face: proportional;
	font-size: large;
	font-style: bold;
	font-color: white;
	background {
		color: darkpink;
	}
	border: none;
	layout: horizontal-center | horizontal-expand;
}

focused {
	padding: 2;
	padding-vertical: 3;
	padding-left: 3;
	padding-right: 3;
	padding-top: 10;
	padding-bottom: 10;
```

```
	background-type: round-rect;
	background-arc: 8;
	background-color: pink;
	border {
		type: round-rect;
		arc: 8;
		color: yellow;
		width: 2;
	}
	font {
		style: bold;
		color: black;
		size: small;
	}
	layout: expand | center;
}

list {
	padding-left: 5;
	padding-right: 5;
	padding-vertical: 10;
	padding-horizontal: 10;
	background {
		color: pink;
		image: url( heart.png );
	}
	columns: 2;
	columns-width: equal;
	layout: horizontal-expand | horizontal-center | vertical-center;
}

listitem {
	margin: 2; /* for the border of the focused style */
	padding: 2;
	padding-vertical: 3;
	padding-left: 3;
	padding-right: 3;
	padding-top: 10;
	padding-bottom: 10;
	background: none;
	font-color: white;
	font-style: bold;
	font-size: small;
	layout: center;
	icon-image: url( %INDEX%icon.png );
	icon-image-align: top;
}
```

The StringItem: Text, Hyperlink or Button

Texts have no specific attributes, but the padding-vertical attribute has a special meaning:

Attribute	*Required*	*Explanation*
padding-vertical	No	The space between the lines when there the text contains line-breaks.

Depending on the appearance mode[6] of the text, either the “p”, the “a” or the “button” selector is used for dynamic styles:

Dynamic Selector	*Explanation*
p	The “p” selector is used for normal texts.
a	The “a” selector is used for hyperlinks.
button	The “button” selector is used for buttons.

See the general explanation of dynamic styles on page 41 for more details.

The IconItem

Icons can only be used directly. Icons support following additional attributes:

Attribute	*Required*	*Explanation*
icon-image	No	The URL of the image, e.g. “`icon-image: url(icon.png);`”. The keyword %INDEX% can be used for adding the position of the icon to the name, e.g. “`icon-image: url(icon%INDEX%.png);`”. The image used for the first icon will be “icon0.png”, the second icon will use the image “icon1.png” and so on.
icon-image-align	No	The position of the image relative to the text. Either “top”, “bottom”, “left” or “right”. Defaults to “left”, meaning that the image will be drawn left of the text.

This example uses the attributes for designing all icons:

```
icon {
      background: none;
      font-color: white;
      font-style: bold;
      icon-image: url( %INDEX%icon.png );
      icon-image-align: top;
}
```

When the icon-item has a “right” image align and the layout is set to “horizontal-expand”, the image will be drawn directly at the right border of the item (with a gap specified by the padding-right attribute). Otherwise the image will be drawn right of the text with the specified horizontal padding.

The ChoiceItem

The ChoiceItem is used in lists and in choice groups. It supports following additional attributes:

6 The appearance mode can be set in the application code with Item.setAppearanceMode(int).

Attribute	*Required*	*Explanation*
icon-image	No	The URL of the image, e.g. "`icon-image: url(icon.png);`". The keyword %INDEX% can be used for adding the position of the item to the name, e.g. "`icon-image: url(icon%INDEX%.png);`". The image used for the first item will be "icon0.png", the second item will use the image "icon1.png" and so on.
icon-image-align	No	The position of the image relative to the text. Either "top", "bottom", "left" or "right". Defaults to "left", meaning that the image will be drawn left of the text.
choice-color	No	The color in which the check- or radio-box will be painted. Defaults to black.
checkbox-selected radiobox-selected	No	The URL of the image for a selected item. This will be used only when the type of the list or of the choice group is either exclusive or multiple. Default is a simple image drawn in the defined choice-color.
checkbox-plain radiobox-plain	No	The URL of the image for a not-selected item. This will be used only when the type of the list or of the choice group is either exclusive or multiple. Default is a simple image drawn in the defined choice-color. When "none" is given, no image will be drawn for not-selected items. Only the image for selected items will be drawn in that case.

Depending on the type of the corresponding list or choice group, different dynamic selectors are used by a choice item:

Type of List or ChoiceGroup	*Selector*
implicit	listitem
exclusive	radiobox
multiple	checkbox
popup	popup

The ChoiceGroup

A choice group contains several choice items. It supports the "focused-style" attribute:

Attribute	*Required*	*Explanation*
focused-style	No	The name of the style for the currently focused item.

Attribute	*Required*	*Explanation*
columns	No	The number of columns. This can be used to layout the items in a table. Defaults to 1 column.
columns-width	No	Either “normal”, “equal” or the width for each column in a comma separated list (e.g. “`columns-width: 60,60,100;`”). Defaults to “normal”, meaning that each column uses as much space as the widest item of that column needs. The “equal” width leads to columns which have all the same width. The explicit list of column-widths results in the usage of those widths.

The Gauge

A Gauge shows a progress indicator. It supports following additional attributes:

Attribute	*Required*	*Explanation*
gauge-image	No	The URL of the image, e.g. “`gauge-image: url (progress.png);`”. When no gauge-width is defined, the width of this image will be used instead.
gauge-color	No	The color of the progress bar. Defaults to blue.
gauge-width	No	The width of the gauge element in pixels. When no width is defined, either the available width or the width of the provided image will be used.
gauge-height	No	The height of the gauge element in pixels. Defaults to 10. When an image is provided, the height of the image will be used.
gauge-mode	No	Either “chunked” or “continuous”. In the continuous mode only the gauge-color will be used, whereas the chunked mode intersects the indicator in chunks. The setting is ignored when an image is provided. Default value is “chunked”.
gauge-gap-color	No	The color of gaps between single chunks. Only used in the “chunked” gauge-mode or when a gauge with an indefinite range is used. In the latter case the provided color will be used to indicate the idle state. Default gap-color is white.
gauge-gap-width	No	The width of gaps in pixels between single chunks. Only used in the “chunked” gauge-mode. Defaults to 3.
gauge-chunk-width	No	The width of the single chunks in the “chunked” gauge-mode.

Attribute	*Required*	*Explanation*
gauge-show-value	No	Either “true” or “false”. Determines whether the current value should be shown. This defaults to true for all definite gauge items.
gauge-value-align	No	Either “left” or “right”. Defines where the current value of the gauge should be displayed. Defaults to “left”, that is left of the actual gauge item.

The following example shows the use of the gauge attributes:

```
.gaugeStyle {
      padding: 2;
      border-color: white;
      gauge-image: url( indicator.png );
      gauge-color: rgb( 86, 165, 255 );
      gauge-width: 60;
      gauge-gap-color: rgb( 38, 95, 158 );
      /* these setting are ignored, since an image is provided:
      gauge-height: 8;
      gauge-mode: chunked;
      gauge-gap-width: 5;
      gauge-chunk-width: 10;
      */
}
```

When the Gauge item is used with an indefinite range, the gauge-gap-color will be used to indicate the idle state. When the “continuous running” state is entered and an image has been specified, the image will “fly” from the left to the right of the indicator.

The TextField

A TextField is used to get user input. Currently the input is done with the use of the native functions, so that special input modes can be used (like T9). The TextField supports following additional attributes:

Attribute	*Required*	*Explanation*
textfield-width	No	The minimum width of the textfield-element in pixels.
textfield-height	No	The minimum height of the textfield-element in pixels. Defaults to the height of the used font.

The following example shows the use of the TextField attributes:

```
.inputStyle {
      padding: 2;
      background-color: white;
      border-color: black;
      textfield-height: 15;
      textfield-width: 40;
}
```

The DateField

A DateField is used to enter date or time information. Currently the input is done with the use of the native functions, so that special input modes can be used (like T9). The DateField supports following additional attributes:

Attribute	*Required*	*Explanation*
datefield-width	No	The minimum width of the textfield-element in pixels.
datefield-height	No	The minimum height of the textfield-element in pixels. Defaults to the height of the used font.

The appearance of the datefield can also be adjusted using the preprocessing variable "polish.DateFormat", which can be defined in the <variables> section of the J2ME Polish task:

- `<variable name="polish.DateFormat" value="us" />` The US standard of MM-DD-YYYY is used, e.g. "12-24-2004".
- `<variable name="polish.DateFormat" value="de" />` The German standard of DD.MM.YYYY is used, e.g. "24.12.2004".
- All other settings: The ISO 8601 standard of YYYY-MM-DD is used, e.g. "2004-12-24".

Appendix

Introduction to Ant

To be done.

Licenses

J2ME Polish is licensed under the GNU General Public License (GPL) as well as several commercial licenses.

GNU GPL

You can use the GPL license for projects, which are licensed under the GNU General Public License without limitations.

More information about the GPL is available at these sites:

- http://www.gnu.org/licenses/gpl.html
- http://www.gnu.org/licenses/gpl-faq.html

Commercial Licenses

If the source code of your mobile applications should not be published under the GNU GPL license, you can use one of the following commercial licenses:

- Single License
  The single license can be used for the creation and distribution of one mobile application.
- Runtime License
  The runtime license can be used for the creation of any number of mobile applications. The drawback is that all applications together can be installed/sold only a 100 hundred times.
- Enterprise License
  The enterprise license allows to create and sell any number of applications.

The pricing and license terms can be obtained at `http://www.j2mepolish.org/licenses.html`.

Contact

ENOUGH SOFTWARE
Robert Virkus
Vor dem Steintor 218
28203 Bremen
Germany

Telephone	+49 – (0)421 – 98 89 131
Fax	+49 – (0)421 – 98 89 132
Mobile	+49 – (0)160 – 77 88 203
E-Mail	Robert.Virkus@enough.de
Web	www.enough.de