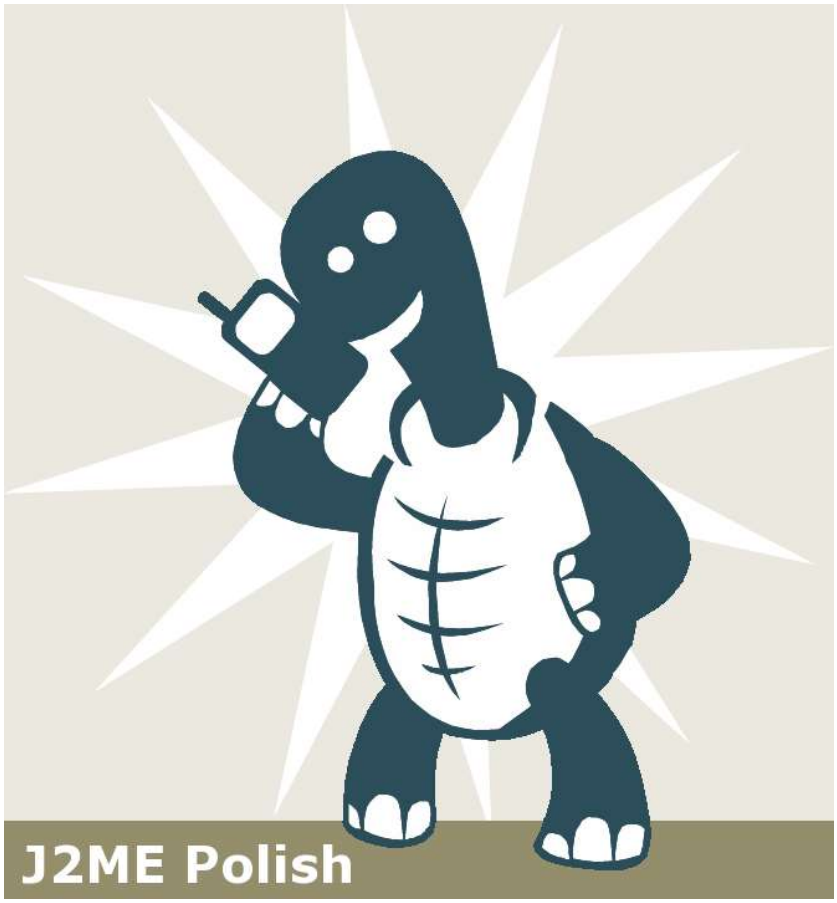
The Complete Guide to J2ME Polish

The Solution for
Professional MIDlet Programming



ENOUGH SOFTWARE

Table of Contents

Preamble....7
Introduction to J2ME Polish....8
Installation....8
An Example Application....8
Building the Application....8
Designing the Application with CSS....9
Debugging and Logging....11
Easy Device-Optimizations with Preprocessing and the Device-Database....12
Fun Fun Fun: The Game-Engine....14
Extending J2ME Polish....14
Installation....15
Requirements....15
Installation of J2ME Polish....15
Updating J2ME Polish....15
Integrating J2ME Polish into Eclipse....15
Integrating J2ME Polish into Borland's JBuilder....17
Integrating J2ME Polish into NetBeans....17
Setup and Configuration of Ant....18
Integrating J2ME Polish with an Existing Project....18
The Device Database....20
vendors.xml....20
groups.xml....20
devices.xml....21
apis.xml....22
Device Capabilities....22
The Build Process....25
Introduction....25
Definition of the J2ME Polish-task....25
Example....25
The info-Section....28
The deviceRequirements-Section....30
The build-Section....32
Attributes of the build-Section....33
Elements of the build-Section....35
<midlet> and <midlets>....35
<obfuscator>....35

The <keep> Subelement....36
The <parameter> Subelement....36
Combining Several Obfuscators....37
<variables>....37
<debug>....38
<jad>....39
Sorting and Filtering JAD attributes....40
<manifestFilter>....41
<preprocessor>....42
Resource Assembling....42
The World of Preprocessing....45
Checking a Single Symbol with #ifdef, #ifndef, #else, #elifdef, #elifndef and #endif....45
Checking Several Symbols and Variables with #if, #else, #elif and #endif....47
Hiding Code....49
Debugging with #debug, #mdebug and #enddebug....49
Debug Levels....50
The Debug Utility Class....50
Using Variables with #=....52
Setting CSS Styles with #style....52
Exclude or Include Complete Files with #condition....54
Defining Temporary Symbols or Variables with #define and #undefine....54
Inserting Code Fragments with #include....55
Analyzing the Preprocessing-Phase with #message....55
Useful Symbols....56
APIs and MIDP-Version....56
J2ME Polish Symbols....56
Useful Variables....56
File Operations....57
The J2ME Polish GUI for Designers....58
A Quick Glance....58
Designing for Specific Devices or Device-Groups....59
The Hierarchy of the “resources” Folder....59
Groups....60
API- and Java-Platform-Groups....60
BitsPerPixel-Groups....61
Dynamic, Static and Predefined Styles....62
Static Styles....62
Predefined Styles....62
Dynamic Styles....63
Extending Styles....65
CSS Syntax....65
Structure of a CSS Declaration....66

Naming....66
Grouping of Attributes....66
Referring to Other Styles....66
Comments....67
Common Design Attributes....67
Structure of polish.css....67
Structure of a Style Definition....67
The CSS Box Model: Margins and Paddings....68
The Layout Attribute....70
Colors....71
Predefined Colors....71
The colors Section....72
How to Define Colors....72
Alpha Blending....73
Fonts....73
Labels....74
Backgrounds and Borders....74
Before and After Attributes....75
Specific Design Attributes....75
Backgrounds....75
Simple Background....75
Round-Rect Background....76
Image Background....77
Circle Background....77
Pulsating Background....78
Pulsating Circle Background....79
Pulsating Circles Background....79
Borders....80
Simple Border....80
Round-Rect Border....80
Shadow Border....81
Circle Border....82
Screens: List, Form and TextBox....82
Predefined Styles for Lists, Forms and TextBoxes....82
Additional Attributes for Screens....83
Dynamic Styles for Screens....84
List....84
Form....84
TextBox....84
Example....84
The StringItem: Text, Hyperlink or Button....87
The IconItem....87
The ChoiceItem....88
The ChoiceGroup....89

The Gauge....89
The TextField....90
The DateField....91
The Ticker....91
The MIDP/2.0-Compatible Game-Engine....93
How It Works....93
Limitations....93
Optimizations....93
Backbuffer Optimization for the TiledLayer....94
Splitting a Image into Single Tiles....94
Defining the Grid Type of a TiledLayer....94
Extending J2ME Polish....96
Integrating a Custom Obfuscator....96
Preparations....96
Creating the Obfuscator Class....96
Integrating the Obfuscator into the build.xml....97
Configuring the Obfuscator with Parameters....97
Creating and Using Custom Preprocessing Directives....97
Preparations....98
Creating the Preprocessor Class....98
Using the registerDirective()-Method....98
Using the processClass()-Method....99
Integrating the Preprocessor into the build.xml....99
Configuring the Preprocessor with Parameters....100
Extending the J2ME Polish GUI....100
Adding a Custom Item By Extending CustomItem....100
Initialisation....100
Interaction Modes....100
Traversal....101
Using CSS for Designing the CustomItem....101
Adding a Custom Background....102
Creating the Client-Side Background-Class....102
Creating the Server-Side Background-Class....104
Integrating the Custom Background....105
Adding a Custom Border....106
Creating the Client-Side Border-Class....106
Creating the Server-Side Border-Class....107
Integrating the Custom Border....108
Contributing Your Extensions....109
Appendix....111
Introduction to Ant....111
Licenses....114

GNU GPL....114
Commercial Licenses....114
Contact....115

Preamble

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, July 2004

Robert Virkus / Enough Software

Introduction to J2ME Polish

This chapter introduces you to some of the possibilities of J2ME Polish. If you want to know specific details, please refer to the following chapters.

Installation

J2ME Polish contains a graphical installer which can be started by double-clicking the downloaded jar-File or by calling "java -jar j2mepolish-[VERSION].jar" from the command line, e.g. "java -jar j2mepolish-1.0.jar".

An Example Application

For this introduction we use a simple application as an example. The application just shows a simple menu which typical for many J2ME applications (listing 1). This is the sample application which is included into the J2ME Polish distribution.

Listing 1: MenuMidlet.java

```
public class MenuMidlet extends MIDlet {

List menuScreen;

public MenuMidlet() {
 super();
 System.out.println("starting MenuMidlet");
 this.menuScreen = new List("J2ME Polish",
List.IMPLICIT);
 this.menuScreen.append("Start game", null);
 this.menuScreen.append("Load game", null);
 this.menuScreen.append("Help", null);
 this.menuScreen.append("Quit", null);
 this.menuScreen.setCommandListener(this);
 System.out.println("intialisation done.");
}

protected void startApp() throws
MIDletStateChangeException {
 System.out.println("setting display.");
 Display.getDisplay(this).setCurrent(
this.menuScreen );
}
[...]
```

J2ME Polish: Overview

Editions: GPL (for Open Source products), Single License (199,-€ valid for one J2ME application), Runtime License (199,-€ valid for an arbitrary number of application which can only be installed a 100 times), Enterprise License (2990,-€ valid for any number of applications). An additional "Developer Seat" license needs to be obtained for every developer (99,- €).

Operating Systems: Windows, Linux

Prerequisites: Java Wireless Toolkit, Ant

Build-Process: preprocessing, device-optimization, resource assembling, device-database, compiling, preverifying, obfuscating, creation of jar and jad files

GUI: Design via CSS, 100% compatible to the MIDP standard, optional.

Game-Engine: MIDP/2.0 game-API for MIDP/1.0 devices.

www.j2mepolish.org

Building the Application

To build the application with J2ME Polish we need to create a file called "build.xml" which controls the build process. This is a standard Ant file, so most IDEs support the editing of this file with syntax highlighting, auto completion etc. Don't worry if you have never worked with Ant before – although it can look quite scary at the beginning it is easy to master after a while. The build.xml file is also included in the J2ME Polish distribution, the main parts of it are shown in listing 2. To build the sample application, we need to right-click the build.xml within the IDE and select "Execute", "Run Ant..." or "make" depending on the IDE. The build process can also be started from the command-line: enter the installation directory and type the "ant" command.

J2ME Polish now preprocesses, compiles, preverifies obfuscates and packages the code automatically for a number of devices. The resulting application bundles can be found in the "dist" folder after J2ME Polish finished the build.

To understand and control the build process, we need to have a closer look at the build.xml (listing 2):

Listing 2: build.xml

```
<project name="example" default="j2mepolish">

<property name="wtk.home" value="C:\wtk2.1">

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

<target name="j2mepolish">
<j2mepolish>
 <info
   name="SimpleMenu"
   version="1.0.0"
   description="A test project"
   jarName="${polish.vendor}-${polish.name}-
example.jar"
   jarUrl="${polish.jarName}"
 </info>
 <deviceRequirements>
  <requirement name="JavaPackage"
               value="nokia-ui" />
 </deviceRequirements>
 <build
   fullscreen="menu"
   usePolishGui="true"
  <midlet class="MenuMidlet" name="Example"/>
 </build>
</j2mepolish>
</target>
</project>
```

At first the J2ME Polish Ant-task needs to be defined with the <taskdef> construct. The Ant-property "wtk.home" points to the installation directory of the Wireless Toolkit.

The J2ME Polish task has the three main-sections "<info>", "<deviceRequirements>" and "<build>". The <info>-element specifies some general information about the project like the name and the version. The <deviceRequirements>-element is used to select the devices for which the application should be builded. The <build>-element is the central controller of the actual build process. This element controls whether the J2ME Polish GUI should be used, what MIDlets are used, which obfuscators should be used, and so on. Here you can also define additional variables and symbols which can be used in the preprocessing step.

Designing the Application with CSS

Even though the application just uses plain MIDP code, we can use the J2ME Polish GUI by setting the "usePolishGui"-attribute of the <build>-element to true (listing 2). J2ME Polish then "weaves" the necessary code and APIs into the application automatically. For specifying or changing the design, the file "polish.css" in the folder "resources" needs to be created or changed (listing 3). As you might have guessed, this file is integrated in the J2ME Polish distribution as well.

The "polish.css" file contains the style definitions for all items and screens of the application. First we define the used colors in the "colors" section. By doing so we can later change the colors of the complete application in one place.

The "title"-style is used for designing the screen-titles. The "focused"- style is used for designing the currently focused item, e.g. of a form or a list.

As we have not used any #style preprocessing directives in the code, we need to use "dynamic" style-selectors like "list" and "listitem" for designing the application. All settings specified in the "list"-style

are used for designing screens of the type “list”. And all settings specified in the “listitem”- style are used for items which are embedded in a list.

All styles support the common attributes margin, padding, layout, font, background and border as well as the special attributes before and after. Most items do also support specific attributes.

The “list”-style in the example utilizes the specific attribute “columns” for specifying that a table with two columns should be used for the list (figure 1).

Fig. 1: The designed sample application.

Designs and resources like images and so on can be adjusted to different devices by placing them into the appropriate subfolder of the “resources” folder. For using a different design for Nokia phones you can add the file “resources/Nokia/polish.css” and for using a special design for the Nokia/6600 phone you can add the file “resources/Nokia/6600”, etc. The same true for other resources like images or sound-files. The resource-assembling does work independently the usage of the J2ME Polish GUI by the way.

When an application has more than one screen and the screens should have different designs, one need use “static” styles. In contrast to dynamic styles, static styles are added during the compilation phase and are, therefore, faster and more memory-efficient. Static styles, however, need changes in the application-code:

Listing 3: polish.css

```
colors {
 bgColor:  rgb(132,143,96);
 brightBgColor:  rgb(238,241,229);
 brightFontColor: rgb(238,241,229);
 fontColor: rgb( 30, 85, 86 );
}

title {
 padding: 2;margin-bottom: 5;
 font-face: proportional; font-size:
 large; font-style: bold; font-color:
 brightFontColor;
 background: none; border: none;
 layout: center | expand;
}

focused {
 padding: 5;
 background {
  type: round-rect; arc: 8; color:
  brightBgColor; border-color:
  fontColor; border-width: 2;
 }
 font {
  style: bold; color: fontColor;
  size: small;
 }
 layout: expand | center;
}

list {
 padding: 5; padding-left: 15; padding-
 right: 15;
 background {
  color: transparent;
  image: url( bg.png );
 }
 layout: expand | center | vertical-
center;
 columns: 2;
}

listitem {
 margin: 2; padding: 5; background:
 none; font-color: fontColor; font-
 style: bold; font-size: small;
 layout: center;
 icon-image: url( icon%INDEX%.png );
 icon-image-align: top;
}
```

is

of

to

For using a static style, the #style-preprocessing-directive is used. This directive can be placed before any constructor of a screen or an item (and some more places like the List-append()-methods) within the code.

```
public MenuMidlet() {
 super();
 System.out.println("starting MenuMidlet");
 //#style mainScreen
 this.menuScreen = new List("J2ME Polish", List.IMPLICIT);
 //#style mainCommand
 this.menuScreen.append("Start game", null);
 //#style mainCommand
 this.menuScreen.append("Load game", null);
 //#style mainCommand
 this.menuScreen.append("Help", null);
 //#style mainCommand
 this.menuScreen.append("Quit", null);
 //#style mainCommand
 this.menuScreen.setCommandListener(this);
 System.out.println("intialisation done.");
}
```

Instead of using the “list” and the “listitem”-styles in the polish.css, you now need to use the styles “mainCommand” and “mainScreen”. Since these are static styles, they need to start with a dot in the polish.css file:

```
.mainScreen {
 padding: 5; padding-left: 15; padding-
 [...]
}

.mainCommand {
 margin: 2; padding: 5; background:
 [...]
}
```

With using the #style-preprocessing directive the code still remains 100% compatible to the MIDP standard, so one use the native/standard GUI for very old devices while using the J2ME Polish GUI for modern devices at the same time.

You can specify in the build.xml file what “resources” folder should be used. This mechanism can be used to create different designs of one application. Figure 2 shows the sample application with a different design. This design is also included in the J2ME Polish distribution and can be used when the “resDir”-Attribute is set to “resources2” in the build.xml.

Fig. 2: The same application with another design. The background is actually animated: it starts white and dithers to the shown pink.

Debugging and Logging

Logging and debugging is complex and difficult task under J2ME. On one hand logging messages need memory as well as computing time and on the other hand there are no logging frameworks like log4j available for J2ME. This is why J2ME Polish includes such a framework with these features:

- Logging-levels like debug, info, warn, error etc. can be defined for single classes or packages.
- User-defined levels can be used, too.

- Logging-messages can be viewed on real devices as well.
- Logging-messages which are not active, will not be compiled and use, therefore, no resources at all.
- The complete logging can be deactivated.
- Simple System.out.println()-calls are used for logging.

The logging is controlled with the <debug> -element in the “build.xml” file :

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

The “enable”-attribute defines whether logging should be activated at all. The “showLogOnError”-attribute can be used for showing the log automatically, whenever an error has been logged. This only works when the GUI of J2ME Polish is used, though. An error can be logged just by appending the error object to a System.out.println()-call:

```
try {
  callComplexMethod();
  //#debug info
  System.out.println( "complex method succeeded." );
} catch (MyException e) {
  //#debug error
  System.out.println( "complex method failed" + e ); // when showLogOnError is true
}                                                    // this will trigger the log
```

When the “verbose”-attribute is true, after each logging message the current time and the source-code file as well line of the message will be added. This eases the location of errors especially when obfuscation is used. The J2ME APIs of J2ME Polish will also specify details of errors whenever they throw an exception, when the “verbose”-attribute is true. The “level“-attribute controls the general logging-level which is used when no specific level has been assigned for a class. The logging-levels are hierarchically ordered:

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

When a class has the logging-level “info” assigned, messages with a level of “warn”, “error” and so on will also be logged, but no “debug” messages.

In the source code logging messages will be introduced with the #debug-preprocessing-directive, which can specify the logging-level, e.g. “//#debug info”. When no level is specified the “debug” level is assumed.

Easy Device-Optimizations with Preprocessing and the Device-Database

Device optimizations are mostly needed for the user interface of an application. The UI of J2ME Polish adapts automatically to devices and can – thanks to the resource assembling – be easily adjusted to different devices and device-groups.

There are however, other situations in which device optimizations are useful. With the help of the preprocessing capabilities and device database of J2ME Polish, these optimizations are done easy and without risking the portability of the application.

Often the preprocessing directives “#ifdef”, “#if” and “#=” are used for optimizations:

When an API is required for a specific functionality, just use the “polish.api.[api-name]” symbol, which is defined for each supported API:

```
//#if polish.api.mmapi
   // this device supports the Mobile Media API
   [...]
//#else
   // this support does not support the mmapi
   [...]
//#endif
```

One can differentiate between MIDP-versions with the “polish.midp1” and the “polish.midp2” symbol:

```
//#if polish.midp2
   // this device supports MIDP/2.0 standard
   [...]
//#else
   // this support does support the MIDP/1.0 standard
   [...]
//#endif
```

To distinguish between the CLDC/1.0 and the CLDC/1.1 standard, the “polish.cldc1.0” and “polish.cldc1.1” symbols can be used:

```
//#if polish.cldc1.1
   float f = 1.2f;
   [...]
//#endif
```

Several requirements can be combined, too:

```
//#if polish.api.mmapi || polish.midp2
   // this device supports at least the basic Mobile Media API
   [...]
//#endif
```

One can compare variables as well:

```
//#if polish.BitsPerPixel >= 12
   // this device uses at least 12 bits per pixel.
//#endif
//#if polish.Vendor == Nokia
   // this is a Nokia device
//#endif
```

Variables can be defined as well and be used within the source code. An URL could be defined in the build.xml like this:

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

The defined variable can be used in the source code with the “#=”-directive:

```
//#ifdef update-url:defined
   //#= public static final String UPDATE_URL = "${ update-url }";
//#else
   // no variable definition was found, use a default-value:
   public static final String UPDATE_URL = "http://default.com/update";
//#endif
```

Fun Fun Fun: The Game-Engine

The game-engine of J2ME Polish lets you use the Game-API of the MIDP/2.0 standard for MIDP/1.0 devices as well. No changes in the source-code are necessary, since the necessary API is weaved into the application automatically for MIDP/1.0 devices.

The complete API can be used: Sprite, GameCanvas, TiledLayer, LayerManager and Layer. The current limitations are that no pixel-level collision-detections and no sprite-transformations for non-Nokia devices are supported (for Nokia devices sprite transformations are supported).

The game-engine can be optimized for several usage scenarios, you can specify that the backbuffer-optimization should be used for the TiledLayer implementation, for example.

Extending J2ME Polish

J2ME Polish can be extended in several ways:

Build-Tools:

- Obfuscator: integrate your own (or a third party) obfuscator
- Preprocessing: add your own preprocessing directives

GUI:

- Backgrounds and Borders: custom borders or backgrounds enhance the possibilities
- Items: extend and the standard CustomItem – and design it using CSS.

How the extensions are made is described in the “Extending J2ME Polish Chapter” on page 96.

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 test j2mepolish" to skip the obfuscation and to include debug-information of 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 contains 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. You should find the sample application now in the Package-Explorer (package de.enough.polish.example).

If the sources have not been integrated automatically, set the source-directory of the project to the “src” directory: Select “Project” -> “Properties...” -> “Java Build Path” -> “Source” and add the source-folder “src” there.

When the classpath was not set correctly, please include the following jar-files from the “import” folder

to your classpath: import/midp2.jar, import/enough-j2mepolish-client.jar, import/mmapi.jar, import/nokia-ui.jar, import/wmapi.jar. Actually only the first two libraries are needed for the sample application, but when the others are included, future enhancements are easier to accomplish.

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".

If the “Run Ant...” command is not shown, select the “build.xml” file, then open the Menu “Run” and select “External Tools” -> “Run as” -> “Ant Build”.

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.

Integrating J2ME Polish into Borland's JBuilder

To integrate J2ME Polish into JBuilder 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 JBuilder and create a new project called "myproject".

In the project-dialog select the appropriate path and confirm the “src” folder as the main source-folder. Switch to the “Required Libraries” tab and select “Add...” and then “New...”. Enter “MIDP-Development” or similar as the name of the library and add the files “enough-j2mepolish-client.jar”, “midp-2.jar”, “mmapi.jar”, “wmapi.jar” and “nokia-ui.jar” to the library path.

Note: only the “midp-2.jar” is actually needed by the sample application, but if you later want to explore the full possibilities of J2ME Polish you already have all important libraries included.

Now create the new project.

After the project has been created, you need to integrate the provided “build.xml” file: Select “Wizard” -> “Ant” -> “Add...” and select the file “build.xml” in the project-root. Now the “build.xml” is shown in the project-view. Important: you need to deactivate the Borland-compiler for building the actual applications: Right-click the “build.xml” file, select “Properties...” and de-select the “Use Borland Java compiler” check box.

You can now build the sample application by right-clicking the “build.xml” file and selecting “Make”. You will find the created J2ME application files in the “dist” folder of your project, after you have switched to the “File Browser” view.

Integrating J2ME Polish into NetBeans

For integrating J2ME Polish with NetBeans mount the directory into which you have installed J2ME Polish, right-click the "build.xml" file and choose "Execute".

- Create a New Project: Go to "Project" -> "Project Manager" and select "New..." and call the new project "j2mepolish".
- Mounting the Filesysten: Select "File" -> "Mount Filesystem..." -> "Local Directory" and select the folder into which you have installed J2ME Polish.
  Note: Don't navigate into the directory. Just select the J2ME Polish directory and click "Finish".
- Executing J2ME Polish: Open the mounted file-system (switch to the file-system-window by pressing "Control-2"), locate the file "build.xml", right-click it and select "Execute". After the execution you will find the created applications in the "dist" folder.
- Mounting the Libraries: Open the mounted file-system, open the "import" directory and right-click the "enough-j2mepolish-client" jar and select "Mount JAR". Do the same with the libraries "midp2", "mmapi", "nokia-ui" and "wmapi".
- Open the Sample Application: You can now change the sample application, if you want to. You can

find it in the "src" directory (in the package "de.enough.polish.example").

Setup and Configuration of Ant

Ant only needs to be configured when it should be called from the command-line. Since Ant is integrated in every modern Java-IDE, you just need to right-click the “build.xml”-file in your IDE and choose “Run Ant...” (or similar).

After you have downloaded and installed Ant (http://ant.apache.org), you need to set your PATH-environment variable, so that the “ant”-command can be found. If you have installed Ant into “C:\tools\ant” then enter following command on your Windows-command-line (or your shell-script):

```
SET PATH=%PATH%;C:\tools\ant\bin
```

You can change the PATH variable permanently in the System-Settings of Windows (Control Center -> System -> Advanced -> Environment variables).

Now you need set the JAVA_HOME variable as well, e.g.:

```
SET JAVA_HOME=C:\j2sdk1.4.2
```

Under Unix/Linux/Mac OS X please use the “export” command instead of the “SET” command.

Now you should be able to issue following calls from the command-line (make sure that you are in the J2ME Polish installation folder):

```
echo Just calling ant to build and obfuscate the example:
ant
echo Now calling ant with the test-property set to true, so the build is faster:
ant test j2mepolish
```

Integrating J2ME Polish with an Existing Project

You can use J2ME Polish for any existing project by copying the file "build.xml" as well as the folders "import" and "resources" to the root-folder of the project. You then need to adjust the "build.xml" and the "resources/polish.css" files.

Here are the required steps:

- copy your sources to the "src" folder of your J2ME Polish installation (or copy the "build.xml", the "import" folder and the "resources" folder to your project-root).
- Adjust the "build.xml" file: you need to specify your MIDlet class in the <midlet> element. You might also want to adjust the <deviceRequirements> element, currently applications are build for 4 device-groups:
  1) Nokia Series 60 ("Nokia/Series60"),
  2) Nokia Series 60 with MIDP/2.0 ("Nokia/Series60Midp2"),
  3) Any MIDP/1.0 phone("Generic/midp1"),
  4) Any MIDP/2.0 phone ("Generic/midp2").

- If you want to use the J2ME Polish GUI, you need to make changes to the "resources/polish.css" file.
  Tip: use dynamic styles like "form" and "list" for a start.
- If you do not want to use the J2ME Polish GUI, disable it by setting the "usePolishGui"-attribute of the <build>-element to "false".
- Ensure that your Ant-setup is correct, do you have set the JAVA_HOME environment variable?

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 preprocessing (compare chapter “The world of preprocessing”, page 45). 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 “Series60” 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 one 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>hasPointerEvents, 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 this calculation.

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.

Device Capabilities

The device database of J2ME Polish supports a number of capabilities, which are useful for preprocessing and the assembling of resources.

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
JavaConfiguration	The supported Java configuration. Currently either CLDC/1.0 or CLDC/1.1.	*Variable:* polish.JavaConfigiration *Symbols:* polish.cldc1.0 or polish.cldc1.1	cldc1.0 or cldc1.1
JavaPackage	Supported APIs, e.g. "nokia-ui, mmapi"	*Variables:* polish.api, polish.JavaPackage *Symbols (example):* polish.api.nokia-ui polish.api.mmapi	Respectively the name of the supported API, e.g. nokia-ui or mmapi (one group for each supported API). Example: nokia-ui, mmapi

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.video.3gpp polish.video.mpeg-4 polish.VideoFormat.3gpp polish.VideoFormat.mpeg-4	One group for each supported video format. Example: 3gpp mpeg-4
SoundFormat	The sound formats which are supported by the device, e.g. “midi, wav”	*Variable:* polish.SoundFormat *Symbols (examples):* polish.audio.midi polish.audio.wav polish.SoundFormat.midi polish.SoundFormat.wav	One group for each supported audio format. Example: midi wav

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 111. 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 need also define where to find the Wireless Toolkit by defining the Ant-property “wtk.home”:

```
<property name="wtk.home" value="C:\WTK2.1" />
```

On a system without the Wireless Toolkit (like Mac OS X) you can still use J2ME Polish by defining the preverify-attribute of the <build>-element instead of using the “wkt.home”-property.

Example

The following example shows a complete build.xml file:

```
<!-- This file controls the build process.                    -->
<!-- The most important target is the j2mepolish-target,      -->
<!-- which controls for what devices the application should   -->
<!-- be created and so on.                                    -->
<!--                                                          -->
<!-- Important: when you have no Wireless Toolkit installed   -->
<!-- you need to define the "preverify"-attribute             -->
<!-- of the <build>-element of the J2ME Polish task.          -->
<!--                                                          -->
```

```
<!-- When you call Ant from the command-line, you can         -->
<!-- call "ant test j2mepolish" to skip the obfuscation       -->
<!-- and to build the example for fewer handsets.            -->
<!-- The default target builds and obfuscates the example.    -->
<!--                                                          -->
<!-- The full documentation can be found at                   -->
<!-- http://www.j2mepolish.org                                -->
<!--                                                          -->
<!-- Have fun!                                                -->
<project
      name="enough-j2mepolish-example"
      default="j2mepolish">

<!-- The wtk.home property should point to the directory      -->
<!-- containing the Wireless Toolkit.                         -->
<property name="wtk.home" value="C:\WTK2.1" />

<!-- Definition of the J2ME Polish task:                      -->
<taskdef name="j2mepolish" classname="de.enough.polish.ant.PolishTask"
classpath="import/enough-j2mepolish-
build.jar:import/jdom.jar:import/proguard.jar:import/retroguard.jar"/>


<!-- 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="test" >
      <property name="test" value="true" />
</target>

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

<!-- In this target the J2ME Polish task is used.             -->
<!-- It has 3 sections:                                       -->
<!--    1. The info-section defines some general information  -->
<!--    2. The deviceRequirements-section chooses the devices -->
<!--       for which the application is optimized.            -->
<!--    3. The build-section controls the actual build        -->
<!--       process.                                           -->
<target name="j2mepolish"
            depends="init"
            description="This is the controller for the J2ME build process."
            >
      <j2mepolish>
          <!-- general settings -->
            <info
             license="GPL"
             name="J2ME Polish"
             version="1.3.4"
             description="A sample project"
             vendorName="Enough Software"
             infoUrl="http://www.j2mepolish.org"
```

```
            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 -->
          <!-- In the test mode the application is build only for the  -->
          <!-- Nokia/3650 and the 6600 phones, otherwise  -->
          <!-- the second deviceRequirements will be used instead. -->
          <deviceRequirements if="test">
                <requirement name="Identifier" value="Nokia/3650, Nokia/6600" />
          </deviceRequirements>
          <deviceRequirements unless="test">
                <requirement name="Identifier" value="Nokia/Series60,
Nokia/Series60Midp2, Generic/midp2, Generic/midp1" />
                <!-- on could use other devices for real builds, e.g. :
                <or>
                      <and>
                            <requirement name="JavaPackage" value="nokia-ui" />
                            <requirement name="BitsPerPixel" value="16+" />
                      </and>
                </or>
                 -->
          </deviceRequirements>
        <!-- build settings -->
          <build
           symbols="ExampleSymbol, AnotherExample"
           imageLoadStrategy="background"
           fullscreen="menu"
           usePolishGui="true"
           resDir="resources"
           >
            <!-- midlets definition -->
            <midlet class="de.enough.polish.example.MenuMidlet" name="Example" />
            <!-- project-wide variables - used for preprocessing  -->
            <variables>
                <variable name="update-url" value="http://www.enough.de/update" />
                <variable name="title" value="J2ME Polish" />
            </variables>
            <!-- obfuscator: don't obfuscate when the test-property is true -->
            <obfuscator unless="test" enable="true" name="ProGuard" />
            <!-- debug settings: only include debug setting when the test-
                 property is true -->
            <debug if="test" enable="true" showLogOnError="true" verbose="true"
level="error">
                <filter pattern="de.enough.polish.example.*" level="debug" />
                <filter pattern="de.enough.polish.ui.*" level="warn" />
            </debug>
            <!-- user defined JAD attributes can also be used: -->
            <jad>
                <attribute name="Nokia-MIDlet-Category" value="Game"
if="polish.Vendor == Nokia" />
            </jad>
```

```
            </build>
      </j2mepolish>
</target>


<target name="clean"
        description="allows a clean build. You should call [ant clean] whenever you
made changes to devices.xml, vendors.xml or groups.xml">
      <delete dir="build" />
      <delete dir="dist" />
</target>
</project>
```

In the first section the J2ME Polish-task and the location of the wireless-toolkit are defined, followed by the build-targets “test”, “init”, “j2mepolish” and “clean”. The targets “test” and “init” are responsible for entering the test mode.

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

You can force a complete rebuild by calling “ant clean”. This can be necessary after you made changes in the device database or in rare cases when there is a compile error in the J2ME Polish packages.

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

info-Attribute	***Required***	***Explanation***
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"`.
permissions	No	The permissions which are needed by this application, e.g. `"javax.microedition.io.Connector.http"`.
optionalPermissions	No	The permissions which are useful for this application to work.

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.
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+" />`

requirement-name	*Explanation*
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+" />
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="hasPointerEvents" />

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”.

build-Attribute	***Required***	***Explanation***
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.
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.
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.

obfuscator-Attribute	*Required*	*Explanation*
name	No	The name of the obfuscator which should be used. Defaults to “ProGuard”. Possible values are at the moment either “ProGuard” or “RetroGuard”.
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 (compare page 96).
classPath	No	The classpath for this obfuscator. This is useful for integrating a third party obfuscator.

The <obfuscator> supports the subelements <keep> and <parameter>.

The <keep> Subelement

The <keep> element 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.

The <parameter> Subelement

The <parameter>-element is used to specify parameters for the obfuscator. This can be useful for integrating third party obfuscators which require additional settings (see page 96):

```
<obfuscator unless="test" enable="true" name="ProGuard" >
      <keep class="com.company.dynamic.SomeDynamicClass" />
      <keep class="com.company.dynamic.AnotherDynamicClass" />
      <parameter name="scriptFile" value="../scripts/obfuscate.script" />
</obfuscator>
```

parameter-Attribute	*Required*	*Explanation*
name	Yes	The name of the parameter.

parameter-Attribute	*Required*	*Explanation*
value	Yes	The value of the parameter.

Combining Several Obfuscators

Several obfuscators can be combined just by specifying several <obfuscator> elements. When the <keep> subelement is used, it only needs to be specified on of the <obfuscator> elements. J2ME Polish will then use the obfuscated output of one obfuscator as input for the following obfuscator.

<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" />
      <variable name="polish.TiledLayer.splitImage"
                value="true"
                if="polish.Vendor == Nokia" />
</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 needs the attributes name and value:

variable-Attribute	*Required*	*Explanation*
name	Yes	The name of the variable.
value	Yes	The value of the variable.
if	No	The Ant-property which needs to be “true” or the preprocessing term which needs result true for this variable to be included.
unless	No	The Ant-property which needs to be “false” or the preprocessing term which needs result false for this variable to be included.

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" showLogOnError="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...");
```

or

```
//#debug error
System.out.println("could not load something: " + e);
```

In the chapter “The world of preprocessing” on page 49 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. When both enable and if are true, then the preprocessing symbol “polish.debugEnabled” will be defined.
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.

debug-Attribute	*Required*	*Explanation*

showLogOnError	No	Either “true” or “false”. When “true” the log containing all logging-messages will be shown when an exception is logged.
		An exception is logged by printing out a message which is followed by a plus and the exception variable:
		`//#debug error`
		`System.out.println("could not load something: " + e);`
		Alternatively the Debug-class of J2ME Polish can be used directly. Then the log will be shown when the
		`Debug.debug( String message, Throwable exception )`
		method is called.
		The log can only be shown automatically, when the J2ME Polish GUI is used.

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.*`”
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.

<jad>

With the <jad>-element user-defined attributes can be added to the JAD[1]-file as well as the MANIFEST-file which is contained in the created JAR-file. The <jad> element contains a number of <attribute>-sub-elements which define the actual attributes:

```
<jad>
 <attribute
      name="Nokia-MIDlet-Category"
```

1 Java Application Descriptor

```
      value="Game"
      if="polish.group.Series40"
      />
 <attribute
      name="MIDlet-Push-1"
      value="socket://:79, com.sun.example.SampleChat, *"
      if="polish.midp2"
      />
</jad>
```

attribute-Attribute	*Required*	*Explanation*
name	Yes	The name of the attribute, e.g. “Nokia-MIDlet-Category”
value	Yes	The value of the attribute.
if	No	The Ant-property which needs to be “true” or the preprocessing term which needs result true for this attribute to be included.
unless	No	The Ant-property which needs to be “false” or the preprocessing term which needs result false for this attribute to be included.

Attributes which are defined in this way can be loaded in the application with the MIDlet.getAppProperty(String name) method, which needs the name of the attribute as an argument and returns the value of that argument. Often it is advisable to use the variable-mechanism of J2ME Polish instead, since such values are actually hard-coded into the application and are, therefore, much faster than the getAppProperty(...)-mechanism.

Sorting and Filtering JAD attributes

The JAD-attributes can filtered and sorted using the <filter> subelement:

```
<jad>
 <attribute
      name="Nokia-MIDlet-Category"
      value="Game"
      if="polish.group.Series40"
      />
 <filter>
      MIDlet-Name, MIDlet-Version,
      MIDlet-Vendor, MIDlet-Jar-URL, MIDlet-Jar-Size,
      MIDlet-Description?, MIDlet-Icon?, MIDlet-Info-URL?,
      MIDlet-Data-Size?, MIDlet-*, *
 </filter>
</jad>
```

The <filter> element contains all allowed attribute names in the desired order and separated by commas. Following syntax is used:

Filter	*Example*	*Explanation*
name	`MIDlet-Name`	The complete name of a required attribute.
name followed by a questionmark	`MIDlet-Icon?`	The complete name of an optional attribute.
beginning of a name followed by a star	`MIDlet-*`	All remaining attributes which names start with the given sequence will be included at this position.
star	`*`	All remaining attributes will be included at this position. A star can only positioned at the end of a filter.

An overview about allowed attributes can be found at http://java.sun.com/j2me/docs/wtk2.0/user_html/Ap_Attributes.html.

<manifestFilter>

The manifest filter element can be used to sort and filter manifest attributes. Please note that the first included attribute should always be the “Manifest-Version”-attribute. Otherwise the application will most likely not work:

```
<manifestFilter>
      Manifest-Version, MIDlet-Name, MIDlet-Version, MIDlet-Vendor,
      MIDlet-1, MIDlet-2?, MIDlet-3?, MIDlet-4?, MIDlet-5?,
      MicroEdition-Profile, MicroEdition-Configuration,
      MIDlet-Description?, MIDlet-Icon?, MIDlet-Info-URL?,
      MIDlet-Data-Size?
</manifestFilter>
```

The <manifestFilter> element contains all allowed attribute names in the desired order and separated by commas. Following syntax is used:

Filter	*Example*	*Explanation*
name	`MIDlet-Name`	The complete name of a required attribute.
name followed by a questionmark	`MIDlet-2?`	The complete name of an optional attribute.
beginning of a name followed by a star	`MIDlet-*`	All remaining attributes which names start with the given sequence will be included at this position.
star	`*`	All remaining attributes will be included at this position. A star can only positioned at the end of a filter.

An overview about allowed attributes can be found at

http://java.sun.com/j2me/docs/wtk2.0/user_html/Ap_Attributes.html.

<preprocessor>

The <preprocessor> element can be used to integrate own or third party preprocessors. This can be used for allowing new preprocessing directives for example (compare page 97).

```
<preprocessor
      class="com.company.preprocess.MyPreprocessor"
      classPath="import/preprocessing.jar"
      >
 <parameter
      name="scriptFile"
      value="../scripts/preprocess.script"
      />
</preprocessor>
```

preprocessor-Attribute	***Required***	***Explanation***
class	Yes	The class of the preprocessor.
classPath	No	The classpath for the preprocessor.
if	No	The name of the Ant-property, which needs to be “true” or “yes”, when this <preprocessor> element should be used.
unless	No	The name of the Ant-property, which needs to be “false” or “no”, when this <preprocessor > element should be used.

The preprocessor can be configured using a number <parameter> subelements. Each <parameter> subelement needs to define the attributes “name” and “value”.

Resource Assembling

During the build process resources like images, sound files etc. are assembled for each device.

Resources for all devices will be placed into the “resources” folder (when no other directory has been specified with the “resDir”-attribute of the <build>-element in the “build.xml” file).

Resources for devices of specific vendors can be placed into the “resources/[vendor-name]” folder, e.g. the “resources/Nokia” folder for Nokia devices. Resources for a specific device can be placed into the “resources/[vendor-name]/[device-name]” folder, e.g. the folder “resources/Nokia/3650” for Nokia/3650 devices.

For more general resources the groups can be used to which a device belongs to. High-color images can be be put into the “resources/BitsPerColor.16+” folder, for example. The groups can be used very efficiently to include only those resources which are actually relevant for a specific device. A device can belong to many different groups, which are defined either implicitly by the capabilities of that

device (compare the section “Device Capabilities” on page 22) or explicitly by setting the <groups>-element of a device in the device-database.

All resources will be copied to the root of the application and more specific resources will overwrite the common resources: If you have a picture named “hello.png” in the “resources” folder as well as in the “resources/Nokia” and in the “resources/Nokia/6600” folder, the version in the “resources” folder will be used for all non-Nokia devices, the version contained in the “resources/Nokia” folder will be used for all Nokia devices, but the image “resources/Nokia/6600/hello.png” will be used for the application which is build for the Nokia/6600 phone. In the application code you can load this image from the root: `Image.createImage( "/hello.png" )`, no matter from where the resource was copied from.

No subfolders can be used within an application, so it is not possible to load resources from a subfolder within the application. Subfolders do add overhead to an application, this is why the usage of subfolders is not allowed.

Instead of

```
InputStream is = getClass().getResourceAsStream("/levels/level.map"); // this is not allowed
```

the following code needs to be used:

```
InputStream is = getClass().getResourceAsStream("/level.map"); // this is ok
```

Subfolders are merely used to differentiate between different devices and device groups.

For a complete list of groups please refer to the section “Device Capabilities” on page 22, the HTML-documentation also lists all groups for each device contained in the device database. Here a couple of examples:

Group	***Type***	***Default Folder***	***Explanation***
midp1	Platform	resources/midp1	When a device supports the MIDP/1.0 platform, resources for this device can be placed into the “resources/midp1” folder.
midp2	Platform	resources/midp2	For devices supporting the MIDP/2.0 platform.
cldc1.0	Configuration	resources/cldc1.0	For devices supporting the CLDC/1.0 configuration.
cldc1.1	Configuration	resources/cldc1.1	For devices supporting the CLDC/1.1 configuration.
mmapi	api	resources/mmapi	When a device supports the Mobile Media API, resources which need this API can be placed into the “resources/mmapi” folder.
motorola-lwt	api	resources/motorola-lwt	For devices which support Motorola's Lightweight Windowing Toolkit-API.

Group	*Type*	*Default Folder*	*Explanation*
nokia-ui	api	resources/nokia-ui	For devices which support Nokia's User Interface API.
wav	audio	resources/wav	For devices which support the WAV sound format.
mp3	audio	resources/mp3	For devices which support the MP3 sound format.
amr	audio	resources/amr	For devices which support the AMR sound format.
midi	audio	resources/midi	For devices which support the MIDI sound format.
mpeg-4	video	resources/mpeg-4	For devices which support the MPEG-4 video format.
h.263	video	resources/h.263	For devices which support the h.263 video format.
3gpp	video	resources/3gpp	For devices which support the 3GPP video format.
BitsPerColor.12+	colors	resources/BitsPerColor.12+	For devices with a color depth of at least 12 bits per color.
BitsPerColor.16	colors	resources/BitsPerColor.16	For devices with a color depth of exactly 16 bits per color.

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 22.

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.

Checking Several Symbols and Variables with #if, #else, #elif and #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.hasPointerEvents || 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.hasPointerEvents || 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`

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

!	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 );
//#enddebug
```

The above example actually results in the same code as:

```
//#debug error
System.out.println("unable to load something: " + e );
```

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 also be used directly for debugging. Especially its showLog()-method can be useful in some circumstances when the “showLogOnError”-attribute is not sufficient.

```
<debug enable="true" showLogOnError="true" verbose="false" level="error">
```

```
      <filter pattern="com.company.package.*" level="info" />
      <filter pattern="com.company.package.MyClass" level="debug" />
</debug>
```

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.debugEnabled
      private Command logCmd = new Command( "show log", Command.SCREEN, 10 );
//#endif
private Screen mainScreen;
[...]
public MyMIDlet() {
      [...]
      //#ifdef polish.debugEnabled
            this.mainScreen.addCommand( this.logCmd );
      //#endif
}
public void commandAction(Command cmd, Displayable screen ) {
      [...]
      //#ifdef polish.debugEnabled
            if (cmd == logCmd) {
                  Debug.showLog( this.display );
            }
      //#endif
}
```

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

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	***Explanation***
Debug.debug(String message)	Same as //#debug System.out.println(message); only that the message will always be logged.
Debug.debug(String message, Throwable exception)	Same as //#debug System.out.println(message + exception); only that the message will always be logged.
Debug.showLog(Display display)	Shows the log containing all messages. When another message is added while the log is shown, the log will be updated.

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 63.

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.

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.

Form constructor	`//#style mainScreen`
	`Form form = new Form( "Menu" );`
	
	`// in subclasses of Form:`
	`//#style mainScreen`
	`super( "Menu" );`
		Use the #style directive before a Form-constructor or before calling super() in subclass-constructors.

Insertion Point	Example	Explanation

List constructor	`//#style mainScreen`
	`List list = new List( "Menu",`
	`List.IMPLICIT );`
	
	`// in subclasses of List:`
	`//#style mainScreen`
	`super( "Menu" , List.IMPLICIT );`
		Use the #style directive before a List-constructor or before calling super() in subclass-constructors.

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 or Variables with #define and #undefine

You can temporarily define and undefine symbols with the #define directive. 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.hasPointerEvents || 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.

You can temporarily define and undefine variables with the #define as well. When the expression after

the #define directive contains the equals-sign, the left part will be taken as the variable name, and the right part as the variable value:

```
//#if !message:defined
      //#define message = Hello world
//#endif
```

You can later just use the defined variable normally:

```
//#= String message = "${ message }";
```

Directive	*Explanation*
//#define [name]= [value]	Defines the specified variable with the given value.
//#undefine [name]	Removes the specified variable.

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).

Analyzing the Preprocessing-Phase with #message

Sometimes the preprocessing itself is rather complex. The #message directive can help to understand the process by printing out messages during the build-process:

```
//#if !basicInput && (polish.hasPointerEvents || polish.mouseSupported)
      //#define tmp.complexInput
      //#message complex input is enabled
//#else
      //#message complex input is disabled
//#endif
```

Within each message any variables can be used. If a variable is not defined, the variable-definition will be included in the message.

```
//#message using the update-url ${update-url}.
```

The above example results in the message "MESSAGE: using the update-url http://update.company.com." being given out, when the variable "update-url" has the value "http://update.company.com". When this variable is not defined, the message "MESSAGE: using the update-url ${update-url}." will be printed out instead.

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`”.

When a device supports the CLDC/1.0 standard, the symbol “`polish.cldc1.0`” is defined. When the CLDC/1.0 standard is supported, the symbol “`polish.cldc1.1`” is defined instead.

J2ME Polish Symbols

Depending on the settings in the build.xml different symbols will be defined. Please compare the “The Build Section”, page 32.

- `polish.usePolishGui` is defined when the GUI of J2ME Polish is used for the current device.
- `polish.debugEnabled` is defined when the logging of messages and the debug-mode is enabled.
- `polish.debugVerbose` is defined when the debug-mode is enabled and is set to “verbose”. When the verbose mode is enabled, thrown exceptions should contain a detailed message.
- `polish.skipArgumentCheck` can be defined in the symbols-attribute of the <build>-element within the “build.xml” file. When this symbol is defined, all GUI classes of J2ME Polish will not check the input variables. This is useful for minimizing the memory footprint when an application is stable.

Useful Variables

Please refer to the section “Capabilities” on page 22 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.[2] 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.[3] 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

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

3 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.

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 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. The resources will copied into the root folder of the application.

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[4]. 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

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

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 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+
18	$2^{18} = 262.144$	BitsPerPixel.18 BitsPerPixel.18+ BitsPerPixel.16+ BitsPerPixel.12+ BitsPerPixel.8+ BitsPerPixel.4+
24	$2^{24} = 16.777.216$	BitsPerPixel.24 BitsPerPixel.24+ BitsPerPixel.18+ 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.
label	This style is used for the menu the label of any item. One can specify another label-style by defining the CSS-attribute “label-style” in the appropriate style, which refers to another style.

Style	*Description*
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[5] in the screen with the style ".startScreen" for example with the following style declaration:

5 StringItems which have the appearance mode Item.HYPERLINK

```
.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.
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.

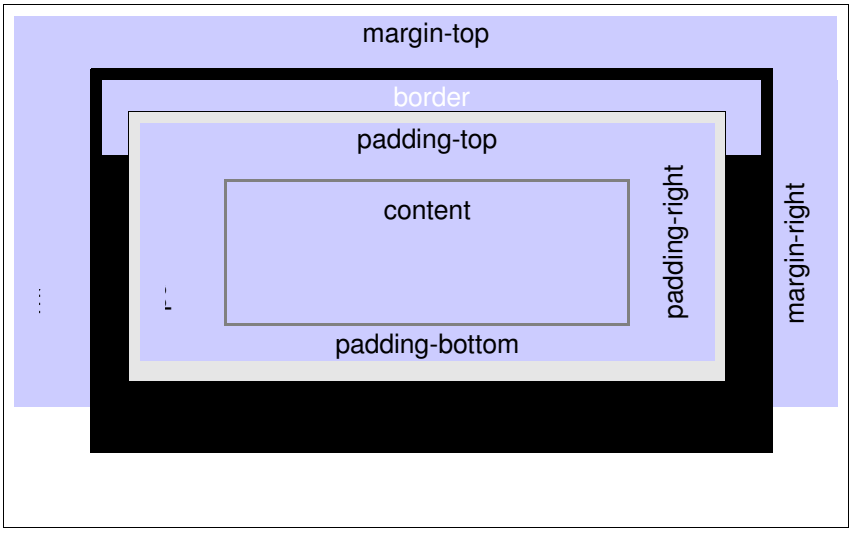
An example of such a complete style definition is the following:

```
/* 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).
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

Many GUI-Items have text elements, which can be designed with the font-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.
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 specification:

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

The same specification can also be written as follows:

```
.myStyle {
      font {
            color: white;
            style: bold;
      }
}
```

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

Labels

All GUI-Items can have a label, which is designed using the pre-defined style “label”. One can specify another-style by using the “label-style”-attribute:

```
.myStyle {
      font {
            color: white;
            style: bold;
      }
      label-style: myLabelStyle;
}

.myLabelStyle {
      font-style: bold;
      font-color: green;
      background-color: yellow;
}
```

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;
}
```

Currently only images can be included.

Fig. 3: The after attribute in action.

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”.
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;
}
```

Circle Background

The circle background paints a circular or elliptical background. It supports following attributes:

Attribute	*Required*	*Explanation*
type	Yes	The type needs to be "circle".

Attribute	*Required*	*Explanation*
color	No	The color of the background, either the name of the color or a direct definition. Defaults to “white”.
diameter	No	With the “diameter” attribute it can be ensured that always a circle and never an ellipse is painted. The diameter then defines the diameter of the circle.

The following example uses a green circle background with a diameter of 20 pixels:

```
.myStyle {
      background-type: circle;
      background-color: green;
      background-diameter: 20;
}
```

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;
```

```
        }
}
```

Pulsating Circle Background

The pulsating circle background paints a circular or background which size constantly increases and decreases. It supports following attributes:

Attribute	*Required*	*Explanation*
type	Yes	The type needs to be “pulsating-circle”.
color	No	The color of the background, either the name of the color or a direct definition. Defaults to “white”.
min-diameter	Yes	The minimum diameter of the circle.
max-diameter	Yes	The maximum diameter of the circle.

The following example uses a green pulsating circle background with a minimum diameter of 20 pixels and a maximum diameter of 40 pixels:

```
.myStyle {
      background-type: pulsating-circle;
      background-color: green;
      background-min-diameter: 20;
      background-max-diameter: 40;
}
```

Pulsating Circles Background

The pulsating circles background paints an animated background of ever-growing circles. It supports following attributes:

Attribute	*Required*	*Explanation*
type	Yes	The type needs to be “pulsating-circles”.
first-color	Yes	The first circle-color, either the name of the color or a direct definition.
second-color	Yes	The second circle-color, either the name of the color or a direct definition.
min-diameter	Yes	The minimum diameter of the circle.
max-diameter	Yes	The maximum diameter of the circle.
circles-number	Yes	The number of circles which should be painted.

Attribute	Required	Explanation
step	No	The number of pixels each circle should grow in each animation phase. Float-values like “1.5” are also allowed. This defaults to “1” pixel.

The following example uses the pulsating circles background with a dark and with a bright color:

```
.myStyle {
      background {
            type: pulsating-circles;
            first-color: bgColor;
            second-color: brightBgColor;
            min-diameter: 0;
            max-diameter: 300;
            circles-number: 8;
            step: 2.5;
      }
}
```

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”.

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.
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: green;
      border-width: 2;
      border-offset: 2;
}
```

Circle Border

The circle border paints a round or elliptical border and supports following attributes:

Attribute	*Required*	*Explanation*
type	Yes	The type needs to be “circle”.
color	No	The color of the border, either the name of the color or a direct definition. Defaults to “black”.
width	No	The width of the border in pixels. Defaults to 1.
stroke-style	No	Either “solid” or “dotted”. Defines the painting style of the border. Defaults to “solid”.

The following example uses a green circle border with a two pixel wide dotted line:

```
.myStyle {
      border-type: circle;
      border-color: green;
      border-width: 2;
      border-stroke-style: dotted;
}
```

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)[6]. 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.

6 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 63).

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[7] 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 63 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;
```

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

```
        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:

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

Type of List or ChoiceGroup	*Selector*
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.
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.

Attribute	*Required*	*Explanation*
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.
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 or handwriting recognition). 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 or handwriting recognition). 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".

The Ticker

A Ticker scrolls a text over the current screen. In J2ME Polish the ticker is situated right above the menu-bar on the bottom of a screen. Following attribute is supported:

Attribute	*Required*	*Explanation*
ticker-step	No	The number of pixels by which the ticker is shifted at each update. Defaults to 2 pixels.

The MIDP/2.0-Compatible Game-Engine

Even though MIDP/2.0 is the future, the market is crowded with MIDP/1.0 devices. J2ME Polish offers a nifty solution for those developers who want to support both platforms with the advanced MIDP/2.0 technology.

All classes of the `javax.microedition.lcdui.game`-package can be used normally:

Class	*Explanation*
GameCanvas	The GameCanvas class provides the basis for a game user interface.
Layer	A Layer is an abstract class representing a visual element of a game.
LayerManager	The LayerManager manages a series of Layers.
Sprite	A Sprite is a basic visual element that can be rendered with one of several frames stored in an Image; different frames can be shown to animate the Sprite.
TiledLayer	A TiledLayer is a visual element composed of a grid of cells that can be filled with a set of tile images.

With this game-engine it is possible to program a game with one source code for MIDP/2.0 as well as MIDP/1.0 devices without changing anything within the source code!

How It Works

The J2ME Polish build-tool identifies the usage of the MIDP/2.0 game API and “weaves” the necessary wrapper classes into the code. The obfuscation step then removes all unnecessary functionality to ensure that only the needed parts are actually included into the application. The J2ME Polish game-engine only needs about 5 to 6 kb extra space in the resulting JAR package, when all classes are used.

Limitations

For technical reasons the collision detection works only with collision rectangles, which should be set tightly for Sprites therefore. A pixel-level collision detection is not supported.

When a class extends the GameCanvas class, it should call the super-implementations, when one of the methods keyPressed(int), keyReleased(int) or keyRepeated(int) is overridden.

The rotation of Sprites is currently only supported for Nokia devices, otherwise the rotation-setting will be ignored.

Optimizations

The TiledLayer implementation has several optional optimizations, which can be triggered by defining specific variables in the “build.xml” file (compare the variables section on page 37).

Backbuffer Optimization for the TiledLayer

The backbuffer optimization uses a buffer, to which the tiles are only painted when they have changed. This can result in a significant higher frames per second throughput, since the complete layer needs to be painted only seldomly.

The drawback of the backbuffer optimization is that it uses more memory and that no transparent tiles can be used. So when memory is an issue or when the tiled layer is not used as the background, it is recommended not to use the backbuffer optimization.

Variable	*Explanation*
polish.TiledLayer.useBackBuffer	Defines whether the backbuffer optimization should be used, needs to be ”true” to enable the optimization.
polish.TiledLayer.TransparentTileColor	The color which is used for tiles which should transparent. This defaults to black.

The following example enables the backbuffer optimization:

```
<variable name="polish.TiledLayer.TransparentTileColor" value="0xD0D000" />
<variable name="polish.TiledLayer.useTileBackBuffer" value="true" />
```

Splitting a Image into Single Tiles

A tiled layer can be drawn significantly faster when the base image is split into single tiles. This optimization needs a bit more memory compared to a basic tiled layer. Also the transparency of tiles is lost when the device does not support the Nokia UI API (all Nokia devices do support this API).

Variable	*Explanation*
polish.TiledLayer.splitImage	Defines whether the base image of a TiledLayer should be split into single tiles. Needs to be “true” to enable this optimization.

The following example enables the single tiles optimization:

```
<variable name="polish.TiledLayer.splitImage" value="true" />
```

Defining the Grid Type of a TiledLayer

The J2ME Polish implementation uses a byte-grid, which significantly decrease the memory footprint, but which also limits the number of different tiles to 128. For cases where more different tiles are used, one can define the type to either “int”, “short” or “byte”.

Variable	*Explanation*
polish.TiledLayer.GridType	Defines the type of the used grid in TiledLayer. This defaults to “byte”. Possible values are “int”, “short” or “byte”.

The following example uses a short-grid, which allows the usage of 32767 different tiles instead of 128 different tiles:

```
<variable name="polish.TiledLayer.GridType" value="short" />
```

Extending J2ME Polish

Integrating a Custom Obfuscator

For integrating another obfuscator, you need to first create a class which extends the de.enough.polish.obfuscate.Obfuscator class. Secondly you need to integrate your obfuscator in the "build.xml" file.

Preparations

Create a new project in your IDE and set the classpath so that it includes the "enough-j2mepolish-build.jar", which can be found in the "import" folder of the installation directory. Also include the "ant.jar" file from your Ant-installation into the classpath.

Creating the Obfuscator Class

Create a new class which extends de.enough.polish.obfuscate.Obfuscator and implement the "obfuscate" method:

```
/**
 * Obfuscates a jar-file for the given device.
 *
 * @param device The J2ME device
 * @param sourceFile The jar-file containing the projects classes
 * @param targetFile The file to which the obfuscated classes should be copied to
 * @param preserve All names of classes which should be preserved,
 *                 that means not renamed or removed.
 * @param bootClassPath A path to the library containing either the MIDP/1.0
 *                 or MIDP/2.0 environment.
 * @throws BuildException when the obfuscation failed
 */
public abstract void obfuscate(  Device device,
                                 File sourceFile,
                                 File targetFile,
                                 String[] preserve,
                                 Path bootClassPath )
throws BuildException;
```

You can make use of following protected variables:

- de.enough.polish.LibraryManager libraryManager: a manager for device-APIs
- java.io.File libDir: the path to the "import" folder
- org.apache.tools.ant.Project project: the Ant project in which the obfuscator is embedded

If you need further configuration settings, you can add them with the <parameter> construct, which is discussed below.

To call the obfuscator, you typically need to accomplish these tasks in the "obfuscate"-method:

1. Set the classpath for the given device: You can get the device-specific APIs by calling `device.getClasspath()` which returns a String array containing the locations of the device-API-

Jars. The usual behavior is to set the classpath for the obfuscator to the given bootClassPath and the device-specific APIs.

2. Specify which classes should be spared from obfuscation: the classes specified by the preserve-array should not be obfuscated at all. These always include the MIDlet classes and maybe some other classes which are loaded with the Class.forName()-mechanism.
3. Start the obfuscator: set the input to the provided sourceFile and the output to the specified targetFile. Both of these files are Jars.

When there is an exception, abort the build by throwing a org.apache.tools.ant.BuildException explaining the details why the obfuscation was aborted.

Integrating the Obfuscator into the build.xml

You can integrate any thirdparty obfuscator with the usual <obfuscator>-element in the build.xml file (compare page 35).

```
<obfuscator unless="test" enable="true"
      class="com.company.obfuscate.MyObfuscator"
      classPath="../obfuscator-project/bin/classes"
      >
      <keep class="com.company.dynamic.SomeDynamicClass" />
      <keep class="com.company.dynamic.AnotherDynamicClass" />
      <parameter name="scriptFile" value="../scripts/obfuscate.script" />
</obfuscator>
```

The “class”-attribute needs to be set to the new class. The classPath-attribute can be used for pointing out the location of the obfuscator. This is only needed when the obfuscator is not on the Ant-classpath.

Configuring the Obfuscator with Parameters

The obfuscator can be configured using <parameter>-subelements in the build.xml. For each <parameter> a corresponding “set[parameter-name]”-method needs to be implemented, which either needs to have one File parameter or one String parameter:

```
      <parameter name="scriptFile" value="../scripts/obfuscate.script" />
```

For the using the above parameter, the obfuscator needs to implement either the public method `setScriptFile( File file )` or the public method `setScriptFile( String value )`. When a method with a File-parameter is used, relative file paths are resolved relative to the location of the build.xml file (or to be more precise relative to the project.getBaseDir() location). The file given as a parameter in the set-method uses an absolute path in that case.

Creating and Using Custom Preprocessing Directives

You can easily use your own preprocessing directives by extending the de.enough.polish.preprocess.CustomPreprocessor class and by integrating your preprocessor with the <preprocessor>-element in the “build.xml” file.

Preparations

Create a new project in your IDE and set the classpath so that it includes the “enough-j2mepolish-build.jar”, which can be found in the “import” folder of the installation directory. Also include the “ant.jar” file from your Ant-installation into the classpath.

Creating the Preprocessor Class

Create a new class which extends de.enough.polish.preprocess.CustomPreprocessor.

Using the registerDirective()-Method

The easiest way to use custom directive is by registering them with the “registerDirective”-method:

```
/**
 * Adds a directive which is searched for in the preprocessed source codes.
 * Whenever the directive is found, the appropriate method
 * process[directive-name] is called.
 * When for example the preprocessing directive "//#hello" should be processed,
 * the subclass needs to implement the method
 * processHello( String line, StringList lines, String className ).
 * <pre>
 * registerDirective("hello");
 * // is the same like
 * registerDirective("//#hello");
 * </pre>
 *
 * @param directive the preprocessing directive which should be found.
 *        The directive needs to contain at least 2 characters (apart from
 *           the beginning "//#").
 *           The "//#" beginning is added when not specified.
 * @throws BuildException when the corresponding method could not be found.
 */
protected void registerDirective( String directive ) throws BuildException
```

When for example the directive “date” is registered, the public method `processDate( String line, de.enough.polish.util.StringList lines, String className )` needs to be implemented.

You can make use of following protected variables:

- de.enough.polish.preprocess.Preprocessor preprocessor: the main preprocessor
- de.enough.polish.preprocess.BooleanEvaluator booleanEvaluator : an evaluator for complex terms which can be used in the #if-directive.

If you need further configuration settings, you can add them with the <parameter> construct, which is discussed below.

In the processing method the line in which the preprocessing directive was found is usually changed. When the line is changed, please make sure that the preprocessing-directive is be removed from the changed line, so that the following preprocessing can continue without errors.

When there is an exception, abort the build by throwing a org.apache.tools.ant.BuildException explaining the details why the preprocessing was aborted.

This example should illustrate the usage of the registerDirective()-method:

```
package com.company.preprocess;

import org.apache.tools.ant.BuildException;
import de.enough.polish.preprocess.CustomPreprocessor;
import de.enough.polish.util.StringList;
import java.util.Date;

public class MyPreprocessor extends CustomPreprocessor {

 public MyPreprocessor() {
       super();
       registerDirective("date");
 }

 public void processDate( String line, StringList lines, String className ) {
       int directiveStart = line.indexOf( "//#date");
       String argument = line.substring( directiveStart + "//#date".length() ).trim();
       int replacePos = argument.indexOf("${date}");
       if (replacePos == -1) {
         throw new BuildException(className + " at line "
              + (lines.getCurrentIndex() + 1)
              + ": Unable to process #date-directive: found no ${date} sequence in line ["
              + line + "." );
       }
       String result = argument.substring(0, replacePos )
              + ( new Date().toString() )
              + argument.substring( replacePos + "${date}".length() );
       lines.setCurrent(result);
 }
}
```

Using the processClass()-Method

For more complex situations, you can override the processClass()-method which allows you to process the whole java-file in one go:

```
       /**
        * Processes the given class.
        *
        * @param lines the source code of the class
        * @param className the name of the class
        */
       public void processClass( StringList lines, String className ) {
              while (lines.next()) {
                     String line = lines.getCurrent();
                     // now process the line...
              }
       }
```

Integrating the Preprocessor into the build.xml

You can integrate your preprocessor with the usual <preprocessor>-element in the build.xml file (compare page 42).

```
<preprocessor
       class="com.company.preprocess.MyPreprocessor"
       classPath="../preprocessor-project/bin/classes"
```

```
      >
      <parameter name="scriptFile" value="../scripts/preprocess.script" />
</preprocessor >
```

The “class”-attribute needs to be set to the new class. The classPath-attribute can be used for pointing out the location of the preprocessor. This is only needed when the preprocessor is not on the Ant-classpath.

Configuring the Preprocessor with Parameters

The preprocessor can be configured using <parameter>-subelements in the build.xml. For each <parameter> a corresponding “set[parameter-name]”-method needs to be implemented, which either needs to have one File parameter or one String parameter:

```
      <parameter name="scriptFile" value="../scripts/obfuscate.script" />
```

For the using the above parameter, the preprocessor needs to implement either the public method `setScriptFile( File file )` or the public method `setScriptFile( String value )`. When a method with a File-parameter is used, relative file paths are resolved relative to the location of the build.xml file (or to be more precise relative to the project.getBaseDir() location). The file given as a parameter in the set-method uses an absolute path in that case.

Extending the J2ME Polish GUI

Adding a Custom Item By Extending CustomItem

You can integrate custom items to J2ME Polish by extending the javax.microedition.lcdui.CustomItem class.

Initialisation

J2ME Polish calls the getPrefContentWidth()-method first with an open height (-1). Then the getPrefContentHeight()-method is called with the actual granted width. When the width had to be adjusted, the custom item will be notified again with the setSize(width. height)-method.

Please note that the Display.getColor(int) and Font.getFont(int) are only available on MIDP/2.0 devices. Such calls should be surrounded by #ifdef-directives:

```
//#ifdef polish.midp2
      Font font = Font.getFont( Font.FONT_STATIC_TEXT );
//#else
      //# Font font = Font.getDefaultFont();
//#endif
```

Interaction Modes

The J2ME Polish implementation supports the interaction modes KEY_PRESS, TRAVERSE_HORIZONTAL and TRAVERSE_VERTICAL.

Traversal

When the custom item gains the focus for the first time, the traverse-method will be called with the CustomItem.NONE direction. Subsequent calls will include the direction (either Canvas.UP, DOWN, LEFT or RIGHT).

Using CSS for Designing the CustomItem

When the CSS-styles should be used for designing the CustomItem, please make sure that the project-classpath contains the “enough-j2mepolish-client.jar”, which can be found in the “import” folder of the J2ME Polish installation directory.

When the J2ME Polish GUI is used, the preprocessing-symbol “polish.usePolishGui” is defined. This can be used when the CustomItem should be used in the J2ME Polish GUI as well in a plain MIDP/2.0 based UI:

```
//#ifdef polish.usePolishGui
      import de.enough.polish.ui.*;
//#endif
```

There are several ways to integrate a CSS-style into the CustomItem:

1. Defining a static style before the super-constructor is called:

```
  public MyItem( String label ) {
    //#style myitem
    super( label );
  }
```

2. Allowing the setting of the style in a second constructor:

```
  //#ifdef polish.usePolishGui
    public MyItem( String label, Style style ) {
      super( label, style );
    }
  //#endif
```

3. Specifying the name of the dynamic style:

```
  //#ifdef polish.useDynamicStyles
     protected String createCssSelector() {
       return "myitem";
     }
  //#endif
```

The second variant is the most flexible one and is encouraged, therefore. When the item MyItem is now used within the code, the appropriate style can specified with the #style directive:

```
//#style coolStyle
MyItem coolItem = new MyItem( label );
```

For reading the specific style settings, the method setStyle(Style) needs to be overridden:

```
//#ifdef polish.usePolishGui
  public void setStyle( Style style ) {
      super.setStyle( style );
```

```
      this.font = style.font;
      this.fontColor = style.fontColor;
      // now read specific attributes:
      //#ifdef polish.css.myitem-color
            String colorStr = style.getProperty( "myitem-color" );
            if (colorStr != null) {
                  this.color = Integer.parseInt( colorStr );
            } else {
                  this.color = 0xFF0000;
            }
      //#else
            this.color = 0xFF0000;
      //#endif
  }
//#endif
```

The setStyle()-method will be called before the preferred content width and height is requested, when the CustomItem has a style associated with it.

Background and border settings will be enforced by J2ME Polish, the CustomItem merely needs to paint its contents.

Adding a Custom Background

For adding a new background, please create a new project called “background-project” and make sure that the files “enough-j2mepolish-build.jar”, “enough-j2mepolish-client.jar” and the “midp2.jar” are on the classpath of the project. These files can be found in the “import” folder of the J2ME Polish installation directory.

Each background needs two implementation classes: one class for the client-side which is responsible for the actual painting and another one for the server-side which is responsible for creating the client-background-class and reading the parameters from the CSS definitions.

As an example we create and integrate an animated background, which paints a “pulsating” circle. Please note that this background is now directly available in J2ME Polish from version 1.0 onwards.

Creating the Client-Side Background-Class

For using a new background, just create a new class which extends the de.enough.polish.ui.Background class.

We are starting with a simple background, which paints a filled out circle. For the constructor we only need one parameter – the desired color of the circle:

```
//#condition polish.usePolishGui

package com.company.backgrounds;

import javax.microedition.lcdui.Graphics;
import de.enough.polish.ui.Background;

public class PulsatingCircleBackground extends Background {

      private final int color;
```

```
	public PulsatingCircleBackground( int color ) {
		super();
		this.color = color;
	}

	public void paint(int x, int y, int width, int height, Graphics g) {
		g.setColor( this.color );
		g.fillArc( x, y, width, height, 0, 360 );
	}

}
```

The paint method is used to render the background and needs to be implemented by all subclasses of Background. The #condition-directive at the top ensures that the PulsatingCircleBackground is included only when the GUI of J2ME Polish is actually used.

This background is useful, but a bit boring, so we add an animation to it: the background-diameter should grow and shrink constantly. For doing this we need to override the animate()-method, which is used to implement animations. When this method returns “true”, a repaint will be triggered resulting in a call of the paint()-method. With returning false the background indicates that no repaint is necessary.

```
public class PulsatingCircleBackground extends Background {

	private final int color;
	private final int maxDiameter;
	private final int minDiameter;
	private int currentDiameter;
	private boolean isGrowing = true;

	public PulsatingCircleBackground( int color, int minDiameter, int maxDiameter ) {
		super();
		this.color = color;
		this.minDiameter = minDiameter;
		this.maxDiameter = maxDiameter;
		this.currentDiameter = minDiameter;
	}

	public void paint(int x, int y, int width, int height, Graphics g) {
		g.setColor( this.color );
		int centerX = x + width / 2;
		int centerY = y + height / 2;
		int offset = this.currentDiameter / 2;
		x = centerX - offset;
		y = centerY - offset;
		g.fillArc( x, y, this.currentDiameter, this.currentDiameter, 0, 360 );
	}

	public boolean animate() {
		if (this.isGrowing) {
			this.currentDiameter++;
			if (this.currentDiameter == this.maxDiameter) {
				this.isGrowing = false;
			}
		} else {
			this.currentDiameter--;
			if (this.currentDiameter == this.minDiameter) {
```

```
                        this.isGrowing = true;
                    }
                }
                return true;
        }
}
```

The implementation uses the field “currentDiameter”, which is either decreased or increased in each run of the animate()-method. The constructor now has two additional parameters: the minimum diameter and the maximum diameter for the circle.

Creating the Server-Side Background-Class

For converting CSS-style information into the appropriate background object, we need to implement a converter class which extends de.enough.polish.preprocess.BackgroundConverter.

This class reads the provided CSS information and creates the source-code for creating a new instance:

```
//#condition false

package com.company.backgrounds;

import java.util.HashMap;
import org.apache.tools.ant.BuildException;
import de.enough.polish.preprocess.BackgroundConverter;
import de.enough.polish.preprocess.Style;
import de.enough.polish.preprocess.StyleSheet;

public class PulsatingCircleBackgroundConverter extends BackgroundConverter {

        public PulsatingCircleBackgroundConverter() {
                super();
        }

        protected String createNewStatement(
                        HashMap background,
                        Style style,
                        StyleSheet styleSheet )
        throws BuildException {
                String minDiameterStr = (String) background.get( "min-diameter");
                if (minDiameterStr == null) {
                        throw new BuildException("Invalid CSS: the pulsating-circle background
needs the attribute [min-diameter].");
                }
                String maxDiameterStr  = (String) background.get( "max-diameter");
                if (maxDiameterStr == null) {
                        throw new BuildException("Invalid CSS: the pulsating-circle background
needs the attribute [max-diameter].");
                }
                // now check if these diameters have valid values:
                int minDiameter = parseInt("min-diameter", minDiameterStr);
                int maxDiameter = parseInt("max-diameter", maxDiameterStr);
                if (maxDiameter <= minDiameter ) {
                        throw new BuildException("Invalid CSS: the [min-diameter] attribute of
the pulsating-circle background needs to be smaller than the [max-diameter].");
                }
                if (minDiameter < 0 ) {
```

```
                    throw new BuildException("Invalid CSS: the [min-diameter] attribute of
the pulsating-circle background needs to be greater or equals 0.");
              }
              // okay, the min- and max-diameter parameters are okay:
              return "new com.company.backgrounds.PulsatingCircleBackground( "
                    + this.color + ", " + minDiameterStr + ", " + maxDiameterStr + ")";
       }

}
```

In the method “createNewStatement” we read the provided values from the HashMap. To parse these values we can the helper-method `parseInt( String attributeName, String attributeValue )`, `parseFloat( String attributeName, String attributeValue )`, `parseColor( String colorValue )` and `getUrl( String url )`.

In our example we want to use the attribute “color”, “min-diameter” and “max-diameter”. The color value is parsed by the super-class already and can be accessed with “`this.color`”. In case no color has been defined, the background color defaults to white. When a required value is missing or a value is set wrong, we throw a BuildException explaining the details of what went wrong. When everything is okay, we return a String in which a new instance of our background is created.

We included the #condition “false”, so that this file is never included in the J2ME application. This is useful when the same source folder is used for the client-background-class as well as the server-background-class.

Integrating the Custom Background

To use our background, we need to ensure that the classes can be found and to use it in the polish.css file:

At first we make sure, that the classpath includes our background-project. The easiest way is to do this in the definition of the J2ME Polish task within the build.xml:

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

Secondly we need to inform J2ME Polish about the additional source-folder.We do this by specifying the “srcDir”-attribute of the <build>-element in the build.xml. Several directories can be specified by separating them with colons:

```
<j2mepolish>
          [...]
      <build
            symbols="ExampleSymbol, AnotherExample"
            imageLoadStrategy="background"
            fullscreen="menu"
            usePolishGui="true"
            srcDir="source/src:../background-project/source/src"
            >
```

```
            [...]
        </build>
</j2mepolish>
```

Thirdly and lastly we need to use the new background in the “resources/polish.css” file of our application. As the type we need to define the SERVER-side converting class:

```
focused {
        padding: 5;
        background {
                type: com.company.backgrounds.PulsatingCircleBackgroundConverter;
                color: white;
                min-diameter: 20;
                max-diameter: 70;
        }
        font {
                style: bold;
                color: fontColor;
                size: small;
        }
        layout: expand | center;
}
```

Now we can build the application by calling Ant to check the result:

Adding a Custom Border

Adding a custom border requires the same steps as for creating a custom background. Instead of extending the Background class, we need to extend the de.enough.polish.ui.Border class. For the server-side the de.enough.polish.preprocess.BorderConverter needs to be extended.

Please note that borders currently do not support the animate()-method, so if you want to use animated borders, you need to implement them with a Background class.

For this example we create a project called “border-project” and add the files “enough-j2mepolish-build.jar”, “enough-j2mepolish-client.jar” and the “midp2.jar” to the classpath of the project. These files can be found in the “import” folder of the J2ME Polish installation directory. Please note that this border is now directly available in J2ME Polish from version 1.0 onwards.

Creating the Client-Side Border-Class

We create a border which should draw a round border around items. The color, width and stroke-style

of the border should be customizable:

```
//#condition polish.usePolishGui

package com.company.borders;

import javax.microedition.lcdui.Graphics;
import de.enough.polish.ui.Border;


public class CircleBorder extends Border {

	private final int strokeStyle;
	private final int color;
	private final int borderWidth;

	public CircleBorder( int color, int width, int strokeStyle ) {
		super();
		this.color = color;
		this.borderWidth = width;
		this.strokeStyle = strokeStyle;
	}

	public void paint(int x, int y, int width, int height, Graphics g) {
		g.setColor( this.color );
		boolean setStrokeStyle = (this.strokeStyle != Graphics.SOLID );
		if (setStrokeStyle) {
			g.setStrokeStyle( this.strokeStyle );
		}
		g.drawArc( x, y, width, height, 0, 360 );
		if (this.borderWidth > 1) {
			int bw = this.borderWidth;
			while (bw > 0) {
				g.drawArc( x + bw, y + bw, width - 2*bw, height - 2*bw, 0, 360 );
				bw--;
			}
		}
		if (setStrokeStyle) {
			g.setStrokeStyle( Graphics.SOLID );
		}
	}

}
```

The #condition-directive ensures that this class is only used when the GUI of J2ME Polish is actually used. In the paint()-method the border is rendered to the screen. When a different stroke-style than Graphics.SOLID is used, we set that style and reset the style in the end of the paint()-method. This is a J2ME Polish specific convention, since the Graphics.DOTTED stroke style is hardly ever used.

Creating the Server-Side Border-Class

The server side border class reads the CSS-attributes from the polish.css file and creates the code for initializing a new border. The class needs to extend the de.enough.polish.preprocess.BorderConverter class:

```
//#condition false
```

```
package com.company.borders;

import java.util.HashMap;
import org.apache.tools.ant.BuildException;
import de.enough.polish.preprocess.BorderConverter;
import de.enough.polish.preprocess.Style;
import de.enough.polish.preprocess.StyleSheet;

public class CircleBorderConverter extends BorderConverter {

	public CircleBorderConverter() {
		super();
	}

	protected String createNewStatement(
				HashMap border,
				Style style,
				StyleSheet styleSheet )
	throws BuildException
	{
		String strokeStyle = (String) border.get("stroke-style");
		String strokeStyleCode;
		if (strokeStyle != null) {
			if ("dotted".equals(strokeStyle)) {
				strokeStyleCode = "javax.microedition.lcdui.Graphics.DOTTED";
			} else {
				strokeStyleCode = "javax.microedition.lcdui.Graphics.SOLID";
			}
		} else {
			strokeStyleCode = "javax.microedition.lcdui.Graphics.SOLID";
		}
		return "new de.enough.polish.extensions.CircleBorder( "
			+ this.color + ", " + this.width + ", " + strokeStyleCode + ")";
	}

}
```

The BorderConverter already parses the color and width of the border. The color defaults to black and the width defaults to 1. So we only need to parse the desired stroke-style. Unless “dotted” is defined, we assume that the Graphics.SOLID stroke-style should be used.

There are some helper methods for parsing the CSS-attributes: `parseInt( String attributeName, String attributeValue ), parseFloat( String attributeName, String attributeValue ), parseColor( String colorValue )` and `getUrl( String url )`.

We included the #condition “false”, so that this file is never included in the J2ME application. This is useful when the same source folder is used for the client-background-class as well as the server-background-class.

Integrating the Custom Border

To use our background, we need to ensure that the classes can be found and to use it in the polish.css file:

At first we make sure, that the classpath includes our border-project. The easiest way is to do this in the

definition of the J2ME Polish task within the build.xml:

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

Secondly we need to inform J2ME Polish about the additional source-folder.We do this by specifying the “srcDir”-attribute of the <build>-element in the build.xml. Several directories can be specified by separating them with colons:

```
<j2mepolish>
          [...]
      <build
            symbols="ExampleSymbol, AnotherExample"
            imageLoadStrategy="background"
            fullscreen="menu"
            usePolishGui="true"
            srcDir="source/src:../border-project/source/src"
            >
          [...]
      </build>
</j2mepolish>
```

Thirdly and lastly we need to use the new background in the “resources/polish.css” file of our application. As the type we need to define the SERVER-side converting class:

```
focused {
      padding: 5;
      background: none;
      border {
            type:
com.company.borders.CircleBackgroundConverter;
            width: 2;
            color: fontColor;
            stroke-style: dotted;
      }
      font {
            style: bold;
            color: fontColor;
            size: small;
      }
      layout: expand | center;
}
```

Now we can build the application by calling Ant to check the result:

Contributing Your Extensions

Please consider to contribute your extensions to the J2ME Polish project. In an open source world, everyone can participate from the work of others.

Do note, however, that J2ME Polish is used commercially as well. Please get in touch with j2mepolish@enough.de to discuss your contribution.

We need the code and a complete documentation of your extension. Preferably the extension is documented in English, but German is also accepted. It is recommended to use the JavaDoc-conventions within the code for the basic documentation. The J2ME Polish project adheres to the normal Java style conventions. For distinguishing local from instance variables, please use the “this” operator for accessing instance variables. This behavior can be enforced in Eclipse by setting the compiler-option “unqualified access to instance variables”.

Appendix

Introduction to Ant

“Apache Ant is a Java-based build tool. In theory, it is kind of like Make, but without Make's wrinkles.”

J2ME Polish uses Ant as its foundation for the build-process, therefore it is useful to get to know Ant a little bit to make the most out of J2ME Polish.

The file “build.xml” is an XML based document which controls and configures Ant. The basic syntax consists of the the <project>-root-element and several <target>-elements. Each <target>-element is normally used to accomplish one or several specific tasks:

```
<project name="enough-polish-example" >

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

</project>
```

In the above example we have one target called “clean” defined which uses two <delete>-tasks for – who would have thought that – deleting two folders.

Assuming that Ant has been set up correctly (compare page 18) and that you are in the directory containing the “build.xml” file, you can call each of the defined targets by calling “ant [target-name]” on the command line. In the above example only one target is available, so you can only call “ant clean”.

Usually there are several targets defined in the “build.xml” file. In that case it us useful to define a default-target, which is executed whenever no target has been specified:

```
<project name="enough-polish-example"
         default="hello"
>

      <target name="hello">
            <echo message="Hello World!" />
      </target>

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

</project>
```

Now the “hello” target will be executed, when just “ant” is called from the command line.

Another powerful feature of Ant are properties, which are like variables with the difference that they can only set once: after a property is set, it can not be changed anymore. This can be combined with dependencies of targets to create complex behavior:

```
<project name="enough-polish-example"
         default="hello"
>
      <target name="deploy">
            <property name="deploy-url" value="http://www.company.com/download"/>
      </target>

      <target name="init">
            <property name="deploy-url" value="http://localhost"/>
      </target>

      <target name="hello" depends="init">
            <echo message="The deploy-url is ${deploy-url}" />
      </target>

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

</project>
```

In the above example the property “deploy-url” is defined in two targets – either in “init” or in “deploy”. When we call “ant hello” or just “ant” the property will be “http://localhost”, because the “hello” target depends on the execution of the “init”-target:

```
> ant
Buildfile: build.xml

init:

hello:
     [echo] The deploy-url is http://localhost

BUILD SUCCESSFUL
Total time: 1 second
```

But when we call “ant deploy hello” the property will contain the value “http://www.company.com/download” instead, since the “deploy”-target will be called first, then the “init”-target (since the “hello”-target depends on the “init”-target) and last but not least the “hello”-target:

```
> ant deploy hello
Buildfile: build.xml

deploy:

init:

hello:
     [echo] The deploy-url is http://www.company.com/download
```

```
BUILD SUCCESSFUL
Total time: 1 second
```

This closes the short introduction to Ant. We recommend either one of the many good books about Ant or to consult the official Ant manual at `http://ant.apache.org/manual/` for further information.

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	j2mepolish@enough.de
Web	www.enough.de