Technical Manual

A830 J2ME™ Developer Guide

CSR00-KJAVA-A830-DGUI-02090309

Version 1.0.2

MOTOROLA
intelligence everywhere™

Table of Contents

1 INTRODUCTION 6

PURPOSE 6
SCOPE 6
DOCUMENT CONVENTIONS 6
DISCLAIMER 7
REFERENCES 8
REVISION HISTORY 8
DEFINITIONS, ACRONYMS, AND ABBREVIATIONS 8
DOCUMENT OVERVIEW 10

2 J2ME™ INTRODUCTION 11

THE JAVA™ 2 PLATFORM, MICRO EDITION (J2ME™) 11
THE MOTOROLA J2ME™ PLATFORM 12
RESOURCES AVAILABLE ON THE MOTOROLA A830 HANDSET 13

3 DEVELOPING AND PACKAGING J2ME™ APPLICATIONS 14

MDP - MOTOROLA DEVELOPER PROGRAM 14
DEVELOPING – TOOLS AND EMULATION ENVIRONMENTS 14
Features to Look For 15
PACKAGING – PUTTING THE PIECES TOGETHER 16
Compiling .java Files to .class Files 16
Preverifying .class Files 16
Creating a Manifest File with J2ME Specific Attributes 16
J2ME Application Naming Convention 17
JARing .class Files and Other Resources 19
Creating the JAD file 19
DOWNLOAD TO DEVICE 21

4 APPLICATION MANAGEMENT 22

MIDLET LIFECYCLE 22
MIDLET SUITE INSTALLATION 23
MIDLET SUITE DE-INSTALLATION 24
MIDLET SUITE UPDATING 24
STARTING, PAUSING, AND EXITING 25
AMS Control of MIDlet State Transitions 25
MIDlet Control of MIDlet State Transitions 29
JAVA SYSTEM 30

5 LIMITED CONNECTED DEVICE USER INTERFACE (LCDUI) 31

OVERVIEW 31
CLASS DESCRIPTION 31
AVAILABLE FONTS 31
Overview 31
Fonts 32

Default Fonts *32*
kJAVA TELEPHONY 32
Functionality *33*
CODE EXAMPLES 33
TIPS 34
CAVEATS 34

6 LIGHTWEIGHT WINDOW TOOLKIT (LWT) 35

OVERVIEW 35
EXAMPLE OF A MIDLET USING LWT PACKAGE 35
CLASS HIERARCHY AND OVERVIEW 37
ComponentScreen *37*
Component *37*
ComponentListener *38*
InteractableComponent *38*
Button *38*
ImageLabel *38*
Checkbox *39*
CheckboxGroup *39*
TextComponent *39*
TextField *40*
TextArea *40*
Slider *40*
FUNDAMENTAL COMPONENT BEHAVIORS 40
Component Management *40*
Component Regions *41*
Component States *42*
Component Layout *43*
Validation Cycle *48*
Focus Management *50*
Key Event Handling *50*
Pointer Event Handling *51*
Rendering *51*
Scrolling *52*
THE COMPONENTSCREEN CLASS 53
ComponentScreen Definition and Constructor *53*
ComponentScreen Methods *53*
THE COMPONENT CLASS 55
Component Definition and Constructor *56*
Component Fields *56*
Component Methods *57*
Using Components *59*
THE COMPONENTLISTENER INTERFACE 60
ComponentListener Interface Definition *60*
ComponentListener Interface Methods *60*
THE INTERACTABLECOMPONENT CLASS 60
InteractableComponent Definition and Constructor *60*
InteractableComponent Methods *61*
THE BUTTON CLASS 62
Button Class Definition and Constructors *62*
Button Class Fields *62*
Button Class Methods *62*
THE IMAGELABEL CLASS 63
ImageLabel Class Definition and Constructors *63*
ImageLabel Class Fields *64*
ImageLabel Class Methods *64*

CHECKBOX CLASS 65
Checkbox Class Definition and Constructors *65*
Checkbox Class Fields *66*
Checkbox Class Methods *66*
Grouping Checkboxes *66*
THE CHECKBOXGROUP CLASS 68
CheckboxGroup Class Definition and Constructor *68*
CheckboxGroup Class Fields *68*
CheckboxGroup Class methods *69*
THE TEXTCOMPONENT CLASS 70
TextComponent Class Definition and Constructor *70*
TextComponent Class Fields *70*
TextComponent Methods *71*
THE TEXTFIELD CLASS 72
TextField Class Definition and Constructor *72*
TextField Class Methods *72*
THE TEXTAREA CLASS 73
TextArea Class Definition and Constructor *73*
TextArea Class Methods *73*
THE SLIDER CLASS 73
Slider Class Definition and Constructor *74*
Slider Class Fields *74*
Slider Class Methods *74*

7 RECORD MANAGEMENT SYSTEM (RMS) 76

OVERVIEW 76
CLASS DESCRIPTION 76
CODE EXAMPLES 76
TIPS 77
CAVEATS 77

8 J2ME NETWORKING 78

OVERVIEW 78
CLASS DESCRIPTIONS 79
HTTP 80
HTTPS 81
TCP SOCKETS 81
SSL SECURE SOCKETS 82
UDP SOCKETS 82
SERIAL PORT ACCESS 82
Communicating on a Port *84*
Example using StreamConnection *84*
IMPLEMENTATION NOTES 85
TIPS 85

9 GAMING 87

FUNCTIONAL DESCRIPTION 87
CLASS HIERARCHY 87
BACKGROUNDMUSIC CLASS 88
BackgroundMusic Methods *88*
Using BackgroundMusic *88*
GAMESCREEN CLASS 88
GameScreen Fields *88*
GameScreen Methods *89*
Using GameScreen *91*
IMAGEUTIL CLASS 92

ImageUtil Fields *92*
ImageUtil Methods *92*
Using ImageUtil *94*
PALLETEIMAGE CLASS 94
PalleteImage Constructor *95*
PalleteImage Methods *95*
Using PalleteImage *96*
PLAYFIELD CLASS 96
Using Static and Animated Tiles *97*
Using Sprites *97*
Defining View Windows *97*
PlayField Constructor *98*
PlayField Methods *99*
Using PlayField *103*
SOUNDEFFECT CLASS 103
SoundEffect Methods *103*
Using SoundEffect *104*
SPRITE CLASS 104
Animation Frames *104*
Sprite Drawing *104*
Sprite Constructor *105*
Sprite Methods *106*
Using Sprite *108*
FILEFORMATNOTSUPPORTEDEXCEPTION 109
FileFormatNotSupportedException Constructors *109*

APPENDIX A: KEY MAPPING OF MOTOROLA A830 HANDSET 110

KEY MAPPING 110

APPENDIX B: HOW TO 112

DOWNLOADING TO THE DEVICE 112
Serial port download procedure *112*
OTA procedure *113*
INSTALLATION 113
STARTING APPLICATIONS 115
EXITING APPLICATIONS 115

APPENDIX C: FREQUENTLY ASKED QUESTIONS 116

APPENDIX D: SUN MICROSYSTEM'S J2ME™ WIRELESS TOOLKIT 119

OVERVIEW 119
CUSTOMIZING THE WIRELESS TOOLKIT TO THE MOTOROLA A830 HANDSET 120
USING STUBBED-OUT CLASSES 121
PACKAGING APPLICATIONS 121

1 Introduction

Purpose

This document describes the application program interfaces used to develop Motorola compliant Java™ 2 Platform, Micro Edition (J2ME™) applications for the A830 handset device, and a description of how to package and deploy those same J2ME applications.

Scope

This document is for all developers involved with the development of J2ME applications for the A830 handset device.

Document Conventions

Convention	Definition
`<install-dir>`	Refers to the directory in which the Motorola SDK components are installed.
|	Either-or choice. [a | b]

Disclaimer

Motorola reserves the right to make changes without notice to any products or services described herein. "Typical" parameters, which may be provided in Motorola Data sheets and/or specifications can and do vary in different applications and actual performance may vary. Customer's technical experts must validate all "Typicals" for each customer application.

Motorola makes no warranty with regard to the products or services contained herein. Implied warranties, including without limitation, the implied warranties of merchantability and fitness for a particular purpose, are given only if specifically required by applicable law. Otherwise, they are specifically excluded.

No warranty is made as to coverage, availability, or grade of service provided by the products or services, whether through a service provider or otherwise.

No warranty is made that the software will meet your requirements or will work in combination with any hardware or applications software products provided by third parties, that the operation of the software products will be uninterrupted or error free, or that all defects in the software products will be corrected.

In no event shall Motorola be liable, whether in contract or tort (including negligence) for any damages resulting form use of a product or service described herein, or for any indirect, incidental, special or consequential damages of any kind, or loss of revenue or profits, loss of business, loss of information or data, or other financial loss arising out of or in connection with the ability or inability to use the Products, to the full extent these damages may be disclaimed by law.

Some states and other jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, or limitation on the length of an implied warranty, so the above limitations or exclusions may not apply to you.

This warranty gives you specific legal rights, and you may also have other rights, which vary from jurisdiction to jurisdiction.

Motorola products or services are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product or service could create a situation where personal injury or death may occur.

Should the buyer purchase or use Motorola products or services for any such unintended or unauthorized application, buyer shall release, indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the designing or manufacture of the product or service.

References

- [1] Sun™ J2ME Documentation, `java.sun.com/j2me/`
- [2] Sun™ MIDP Specification, `java.sun.com/products/midp/`
- [3] Lightweight Window Toolkit Programmer's Guide, `www.motorola.com/developers/wireless/`
- [4] J2ME Wireless Toolkit homepage, `java.sun.com/products/j2mewtoolkit/download.html`
- [5] Motorola Developer Program site, `www.motorola.com/developers/wireless/`

Revision History

Version	Date	Name	Reason
01.00	Nov 04, 2002	CESAR	Final draft
01.01	Nov 18, 2002	MW MDP	Reformatted
01.02	Nov 22, 2002	MW MDP	Revised on review

Definitions, Acronyms, and Abbreviations

Acronym	Description
AMS	Application Management Software
API	Application Program Interface.
CLDC	Connected Limited Device Configuration
FDI	Flash Data Integrator. The memory used to store the applications.
GPS	Global Positioning System
IDE	Integrated Development Environment
ITU	International Telecommunication Union
JAD	Java Application Descriptor
JAID	Java Application Installer/De-Installer
JAL	Java Application Loader

Acronym	Description
JAR	Java Archive. Used by J2ME applications for compression and packaging.
J2ME	Java 2 Micro Edition
JSR 120	Java Specification Request 120 defines a set of optional APIs that provides standard access to wireless communication resources.
JVM	Java Virtual Machine
KVM	KJava Virtual Machine
LCC	Licensee Close Classes
LWT	Lightweight Window Toolkit
MDP	Motorola Developers Program
MIDP	Mobile Information Device Profile
OEM	Original Equipment Manufacturer
OTA	Over The Air
RMS	Record Management System
RTOS	Real Time Operating System
SC	Service Center
SDK	Software Development Kit
SMS	Short Message Service
SU	Subscribe Unit
UI	User Interface
URI	Location Services Unified Resource Identifier
VM	Virtual Machine

Document Overview

This document is organized in the following sections:

- **Section 1** – Introduction**:** this section has general information about this document. It includes document purpose, scope, references, some definitions and location.
- **Section 2** – J2ME Introduction: this section describes some tips of J2ME platform, and the available resources on the Motorola A830 handset.
- **Section 3** – Developing and Packaging J2ME Applications: this section describes some important features to look for when selecting tools and emulation environments. It also describes how to package a J2ME application, and generate JAR and JAD files properly.
- **Section 4** – Application Management: this section describes the lifecycle, installation/de-installation and updating process for a MIDlet suite. It is also described the MIDlet state machine and how to use the Java System feature option on A830.
- **Section 5** – LCDUI: this section describes the Limited Connected Device User Interface API, including the KJava Telephony API.
- **Section 6** – LWT: this section describes the Lightweight Window Toolkit API.
- **Section 7** – RMS: this section describes the Record Management System API.
- **Section 8** – Networking: this section describes the Java Networking API.
- **Section 9** – Gaming: this section describes the Gaming API.
- **Appendix A** – Key Mapping of Motorola A830 handset: this section describes the key mapping of Motorola A830 handset including the key name, key code and game action of all Motorola A830 keys.
- **Appendix B** – How to: this section describes the downloading, installing, removing, starting and exiting of a MIDlet using the Motorola A830 resources.
- **Appendix C** – FAQ: this section describes the more frequently asked questions about using of MIDlet suite applications on Motorola A830 handsets.
- **Appendix D** – Sun's J2ME wireless toolkit: this section describes briefly the Sun's J2ME wireless toolkit documentation. It also describes how to use the stubbed-out classes.

2
J2ME™ Introduction

The Motorola A830 handset includes the Java™ 2 Platform, Micro Edition, also known as the J2ME platform. The J2ME platform enables developers to easily create a variety of Java applications ranging from business applications to games. Prior to its inclusion, services, or applications, residing on small, consumer devices like cell phones could not be upgraded or added without significant effort. By implementing the J2ME platform on devices like the Motorola A830 handset, service providers as well as customers can easily add and remove applications, allowing for quick and easy personalization of each device. This section of the guide provides a quick overview of the J2ME environment and the tools that can used to develop applications for the Motorola A830 handset.

The Java™ 2 Platform, Micro Edition (J2ME™)

The J2ME platform is a new, very small application environment. It is a framework for the deployment and use of Java technology in small devices such as cell phones and pagers. It includes a set of APIs and a virtual machine that is designed in a modular fashion allowing for scalability among a wide range of devices.

The J2ME architecture contains three layers consisting of the Java Virtual Machine, a Configuration Layer, and a Profile Layer. The Virtual Machine (VM) supports the Configuration Layer by providing an interface to the host operating system. Above the VM is the Configuration Layer, which can be thought of as the lowest common denominator of the Java Platform available across devices of the same “horizontal market.” Built upon this Configuration Layer is the Profile Layer, typically encompassing the presentation layer of the Java Platform.

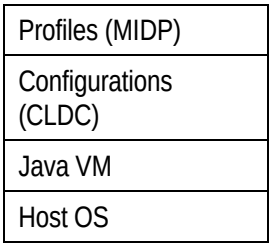

Figure 1. The J2ME Platform Architecture

The Configuration Layer used in the Motorola A830 handset is the Connected Limited Device Configuration 1.0 (CLDC 1.0) and the Profile Layer used is the Mobile Information

Device Profile 1.0 (MIDP 1.0). Together, the CLDC and MIDP provide common APIs for I/O, simple math functionality, UI, and more.

For more information on J2ME, see the Sun™ J2ME documentation.

The Motorola J2ME™ Platform

Functionality not covered by the CLDC and MIDP APIs is left for individual OEMs to implement and support. By adding to the standard APIs, manufacturers can allow developers to access and take advantage of the unique functionality of their devices.

The Motorola A830 handset contains OEM APIs for a wide variety of extended functionality ranging from enhanced UI to advanced data security. While the Motorola A830 handset can run any application written in standard MIDP, it can also run applications that take advantage of the unique functionality provided by these APIs. These OEM APIs are described in this document.

Resources Available on the Motorola A830 handset

The Motorola A830 handset allows access to a richer set of resources than our previous Java™ capable phones. The changes range from a larger heap for Java applications to the presence of a color display. All of the enhancements allow for more compelling and advanced Java applications to be created. In addition to increasing resources present on the device, new APIs to access other device resources were added. These new APIs allow a Java application to leverage other capabilities of the device that are currently not accessible through standard MIDP and CLDC APIs.

Display	
Resolution	176 x 220
Color Depth	12 bit color (4096 colors)
Networking	
Max HTTP, UDP and TCP Socket connections*	4 with any combinations
File & RMS	
Max number of Files/RMS*	500
Java VM	
Heap Size	512 KB
Program Space	1.2 MB
Max Resource Space*	450 KB
Recommended Maximum JAR Size	100 KB

*: These resources are shared with the rest of the phone and there could be less available to Java at any given time.

3
Developing and Packaging J2ME™ Applications

MDP - Motorola Developer Program

Motorola developed the MDP to support developers in their efforts to create and market wireless applications. The program provides access to a wealth of services and support, including assistance in getting application ideas to market, training classes that range from overviews to in-depth courses, instantly accessible online technical support, application certification and industry-leading development software.

A tiered-membership structure enables Motorola Developer Program members to join the program at whatever level is appropriate for them, based on the level of services and support they require. Developers in the early stages of application development, for example, can take advantage of basic development assistance in the form of online downloads and training materials. Those with fully mature applications, on the other hand, can join at a higher level that provides opportunities for certification, exposure to potential markets and other later-stage benefits.

For more information, see the Motorola Developer Program site at `www.motorola.com/developers/wireless`.

Developing – Tools and Emulation Environments

In order to develop applications for a J2ME enabled device, a developer needs some specialized tools to improve development time and prepare the application for distribution. There are several tools available, so this overview is included to help enable developers to make an informed decision on these tools.

Features to Look For

Numerous tools for developing J2ME applications are readily and freely available on the market. Some of their functionalities include:

Class Libraries

J2ME tools include class files for the standard CLDC/MIDP specifications and may also contain class files required to compile device specific code.

One of the main issues with the MIDP 1.0 standard is the lack of device specific functionality. As a stopgap solution, many MIDP 1.0 device manufacturers have implemented Licensee Open Classes that provide the features requested by developers. In order to take advantage of these APIs, choose an SDK that natively supports them or one that can be upgraded to support them.

API Documentation

In addition to providing the class files, most SDKs include reference documentation for the supported APIs. These documents, typically found in either HTML or PDF format, cover the standard CLDC/MIDP specifications as well as the device specific APIs.

Emulation Environment

Although not an absolute necessity if the device is available, most toolkits provide this functionality for multiple devices. The main benefits of an emulation environment are the reduction in development time as well as the ability to develop for devices not yet on the market. The extent to which the toolkits emulate the device can vary greatly.

An emulation environment that accurately reflects how the application will look and feel on the target device, will reduce both your development time and your frustration level. If most of the development is going to take place on the device, then this may not be a big consideration, but if access to the target device is limited or unavailable, accurate emulation is a must. Look for accuracy in the font representation, display dimensions, and pixel aspect ratio, as many wireless devices do not have square pixels.

Along the same lines as accurate look and feel, the tool should also provide accurate performance emulation. A comprehensive tool should provide individual adjustments for performance aspects such as network throughput, network latency, persistent file system access time, and graphics performance. Ideally, these attributes should not only match the target device, but also have the ability to be manually adjusted.

Application Packaging Utility

Most SDKs automatically package the application for deployment onto the target device. Although many tools include this feature, flexibility varies widely. Look for a tool that generates both the manifest and JAD files with the required tags as well as custom tags. The packaging steps required to deploy an application on the Motorola A830 handset are described in a subsequent section.

Is the SDK free?

There are numerous feature rich SDKs freely available on the market. These are not disabled or time limited versions but rather full-blown development environments and tools.

Packaging – Putting the Pieces Together

Once an application has been tested on an emulator and is ready for testing on the actual device, the next step is to package the application and associated components into a JAD/JAR file pair. The files contain both the MIDlet's executable byte code along with the required resources. Although this process is automatically performed by most SDKs and IDEs supporting J2ME, the steps are explained and outlined here.

Compiling .java Files to .class Files

Compiling a J2ME application is no different than any other J2SE/J2EE application. By adding the CLDC/MIDP files (whether functional or stubbed out) to the classpath, any standard Java compiler that is JDK1.3 compliant or greater is sufficient to produce .class files suitable for the preverification step.

Preverifying .class Files

Class files destined for the KVM must undergo a modified verification step before deployment to the actual device. In the standard JVM found in J2SE, the class verifier is responsible for rejecting invalid classes, classes that are not compatible, or have been modified manually. Since this verification step is processor and time intensive, it is not ideal to perform verification on the device. In order to preserve the low-level security model offered by the standard JVM, the bulk of the verification step is performed on a desktop/workstation before loading the class files onto the device. This step is known as preverification.

During the preverification step, the class file is analyzed and a stack map is appended to the front of the file. Although this may increase the class file size by approximately 5%, it is necessary to ensure the class file is still valid when it reaches the target device. The standard J2SE class verifier ignores these attributes, so they are still valid J2SE classes.

Creating a Manifest File with J2ME Specific Attributes

In addition to the class files, a manifest file for the HelloWorld MIDlet needs to be created. Although most J2ME tools will auto generate the manifest file, it can also be created manually using a plain text editor. The following is a sample manifest file:

```
MIDlet-Name: HelloWorld
MIDlet-Version: 1.0.0
MIDlet-Vendor: Motorola, Inc.
MIDlet-1: HelloWorld, ,
com.motorola.midlets.helloworld.HelloWorld
MicroEdition-Profile: MIDP-1.0
MicroEdition-Configuration: CLDC-1.0
```

The device's AMS uses the manifest file to determine the number of MIDlets present within the suite as well as the entry point to each MIDlet. Additionally, the manifest files may contain optional tags that are accessible by the MIDlets within the MIDlet suite. For more information, refer to the MIDP 1.0a specification .

Notes:

The following attributes are mandatory and must be duplicated in both the JAR file manifest and the JAD file. If the attributes are not identical, the application will not install.

- MIDlet-Name
- MIDlet-Version
- MIDlet-Vendor

The manifest contains MIDlet-<n> arbitrary attributes each describing a MIDlet in an application suite.

The MIDlet-1 attribute contains three comma separated fields, the application name, the application icon, and the application class file (entry point). The name is displayed in the AMS user interface to represent the nth application. The application icon is currently not used on the Motorola A830 handset and thus is ignored. To omit the icon field, simply place a <space> in place of the icon name. The application class file is the class extending the `javax.microedition.midlet.MIDlet` class for the nth MIDlet in the suite.

The manifest file is case sensitive.

The manifest must be saved in a file called MANIFEST.MF (case sensitive) within the meta-inf directory.

J2ME Application Naming Convention

According to Motorola MDP procedures for software development teams, you should follow this file naming convention for applications:

```
<App_name>_<Phone_name>_<Type_Demo_or_Full>_<Languages_supp
orted>_VER_<version_number>
```

- **Application name:** Name of application without any specific designation
- **Phone Name:** Product name that this application is designed for: E.g. A830
- **Type_ Demo_or_Full:** D for Demo version, F for Full version.
- **Languages supported:** Append list of languages abbreviated by the ISO naming convention in two letter words.

  Example:

  ```
  TETRIS_A830_D_FREN_VER_01_01_02.JAR
  SNOOD_A830_F_DEEN_VER_03_03_01.JAD
  ```

Notes:

- The Maximum length of MIDlet suite file name is 32 characters, not including the file extension.

- There is a limitation to length of JAD/JAR including URL used in OTA (Over-The-Air Download) of the application. The maximum character length is 256 (JAR file name characters plus OTA site link characters). So, for example:

  ```
  MIDlet-jar-URL: http://www.handango.com/entertainment/
  applicatons/720/TESTAPP_T720_F_DEEN_VER_01_01_01.JAR
  ```

  The MIDlet-Jar-URL in this example has 90 characters, which is less than 256 and is acceptable.

- Language Notation – For each language, appended to each other. e.g. FREN == French , English. Refer to `http://www.geo-guide.de/nfo/tools/languagecode.html` for codes.

- Language Exceptions – No distinctions exist in the ISO guidelines for a few of the languages. They are defined here:
  - Canadian French - CF (Regular French - 'FR');
  - Complex Chinese - CC (Simple Chinese - ZH);
  - Brazilian Portuguese - BP (Regular "Portuguese - PT)
  - Latin American (Columbian) Spanish - LS (Regular Spanish – ES)

- J2ME Application versioning convention –

- Version number: xx.yy.zz.
  - xx - This is commonly referred to as Major revision number;
  - yy – This is commonly referred to as Minor revision number;
  - zz – This is commonly referred to as Build number

- For Preloaded (Demonstration or “Demo”) application version numbering to be:
  - xx = 1 through (and including) 4 (first “x” is lead with a “0”, i.e., xx = 01, 02, 03, or 04)

- For OTA application version (Full version) numbering to be:
  - xx = 5 through (and including) 9 (first “x” is lead with a “0”, i.e., xx = 05, 06, 07, 08, 09)
  - The leading “0” as in the above example (01.02.02) is required;
  - J2ME Application versioning convention: The complete name (all characters) shall be in uppercase letters;
  - Application shall end with .JAD or .JAR;
  - Underscore ("_") must be used to separate the components of the file name, including the version. Periods/Dots (except for those used to indicate the file extension, i.e., “.jad”) cause issues when posted to host OTA site.

JARing .class Files and Other Resources

Once the application is ready to be packaged for the device, its class files and associated resources must be bundled in a Java Archive (JAR) file. The JAR file format enables a developer to bundle multiple class files and auxiliary resources into a single compressed file format. The JAR file format provides the following benefits to the developer and end-user:

- **Portability** – The file format is platform independent.
- **Package Sealing** – All classes in a package must be found in the same JAR file.
- **Compression** – Files in the JAR may be compressed, reducing the amount of storage space required. Additionally, the download time of an application or application suite is reduced.

MIDlet suite cannot be installed successfully if the JAR content has:

- Corrupted JAR file (extraction/decompression failure);
- Corrupted .class file (verification failure);
- .class file bigger than 20k ;
- Resources greater than 20K each;
- Total resources for MIDlet suite greater than 64K.

Creating the JAD file

Although the Java Application Descriptor (also known as an Application Descriptor File) is optional in the MIDP 1.0a specification , J2ME applications targeted for Motorola devices must include a JAD/JAR pair. The following is a sample JAD file for a simple HelloWorld application.

Required:

```
MIDlet-Name: HelloWorld
MIDlet-Version: 1.0.0
MIDlet-Vendor: Motorola, Inc.
MIDlet-Jar-URL: http://www.motorola.com
MIDlet-Jar-Size: 1939
MIDlet-Language*:  EN
Mot-Program-Space-Requirement:
Mot-Data-Space-Requirement:
```

Optional:

```
MIDlet-Description: A sample HelloWorld application.
MIDlet-Info-URL:
MIDlet-Data-Size:
```

The JAD file may be created with any text editor and saved with the same file name prefix as the JAR file. The mandatory MIDlet-Name, MIDlet-Version, MIDlet-Vendor must be duplicated from the JAR file manifest. JAR files containing manifests that do not match the

JAD file will not be installed. The MIDlet-Jar-URL attribute must be described properly; otherwise the MIDlet suite cannot be installed successfully.

Mot-Program-Space-Requirement defines the space required by the application code once the JAR file is installed on the device.

Mot-Data-Space-Requirement defines the data space required by the application that includes resources, application descriptors, and images once the JAR is installed on the device. These are attributes in the JAD file that indicate space required in kilobytes for the application installation. The idea is to use these custom fields to specify the actual amount of memory required when the application is installed on a specific device. When these attributes are present in the JAD file, the device will use this information to check against the available memory to determine if there is sufficient memory for a successful installation.

When using these attributes, developers should first install the application on the actual device to get these values. The installed memory sizes are accessed by selection details on the application menu. Once the actual values are available, the custom fields can be entered in the JAD file to enable successful installation.

Notes:

- The file names of the JAD and JAR are not required to be identical.
- The JAD file is case sensitive. All required attributes in the JAD file must start with "MIDlet-", followed by the attribute name.
- The total file name length is limited to 32 characters, excluding the .jad and .jar extensions. For example, HELWD_A830_F_DEEN_VER_03_03_01.jad occupies 30 characters.
- The MIDlet-Jar-Size must contain the accurate size of the associated JAR file. The number is in bytes.
- It's also important to note that these fields must have associated values with them. Example: "MIDlet-Name: " is not valid but "MIDlet-Name: Snake" is valid.
- One other note, in the "MIDlet-Name", if the MIDlet is a demonstration version (demo version) it should indicate that here. For example, "Breakout" would indicate that the application is a full application. "Breakout Demo" would indicate a demonstration version of the game.
- The new Space fields (Mot-Program-Space-Requirement and Mot-Data-Space-Requirement) should be listed in KB (but without the alpha characters). In addition, regardless of the fraction, the value should always be rounded up. So for instance, if the Mot-Program-Space-Requirement is found to be 124.1 KB it should be listed in the JAD file as 125.
- The MIDlet-Language field: ISO codes, see J2ME Application Naming Convention for details. This field is for informational purposes to verify naming convention of the JAD/JAR files. It does not invoke the language used on the phone, etc. This information should reside in the JAR file.

One other note, for developers working on "Preloaded" (also known as "Demonstration" or "Demo" versions) versions of applications: the "MIDlet-Name" in the JAD file must be identical in both the Preload and Full versions of the application so that the OTA (over-the-air) overwrite/download will work properly.

For more information regarding the JAD file, please refer to the MIDP 1.0a specification .

Download to Device

After creating the JAR and JAD files, the MIDlet can be downloaded to A830 device through Motorola WAP Browser, or through PC. Motorola distributes MIDway tool (see Downloading to the Device) to download MIDlets through PC, and it is available at: `www.motorola.com/develoeprs/wireless/`.

The download using Motorola WAP Browser does not require any additional software resources.

4
Application Management

MIDlet Lifecycle

A MIDlet's lifecycle begins once its MIDlet Suite is downloaded to the device. From that point, the Application Management Software (AMS) manages the MIDlet Suite and its MIDlets. The user's primary user interface for the AMS is the Java Apps feature built into the device's firmware.

From the Games & Apps feature, the user can see each MIDlet Suite on the device. If a MIDlet Suite has only a single MIDlet, then the MIDlet's name is displayed in the Games & Apps menu for that MIDlet Suite. Otherwise, the MIDlet Suite name is displayed. Then when that MIDlet Suite is highlighted, the user has the option of opening the MIDlet Suite and viewing the MIDlets in that MIDlet Suite.

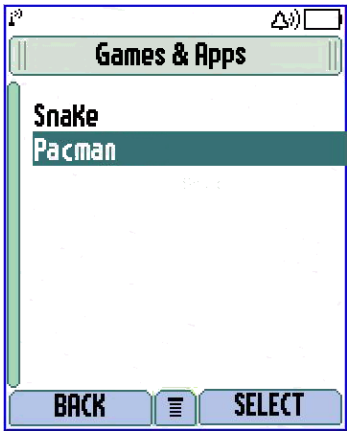

Figure 2. The Games & Apps Menu

From the Games & Apps menu, the user can highlight a MIDlet Suite, selecting the Menu soft key, and bring up the Details dialog for that MIDlet Suite. The Details dialog contains:

- MIDlet Suite Name
- MIDlet Suite Vendor
- MIDlet Suite Version

- The number of MIDlets in the MIDlet Suite
- The Data Space (MIDlet suite resources).
- Program Space (Unpacked JAR)

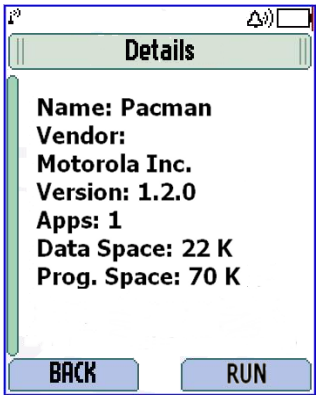

Figure 3. Details Properties for a MIDlet

MIDlet Suite Installation

From the Java Tools menu, the user can install MIDlet Suites. The Figure 4 shows the Java Tools screen. A MIDlet Suite must be installed before any of its MIDlets can be executed. Installation involves extracting the classes from the JAR file and creating an image that will be placed into Program Space. The resources are then extracted from the JAR file and placed into Data Space. The JAR file is then removed from the device, thus freeing up some Data Space where it was originally downloaded.

The space savings from removing the JAR file is one advantage of installation. However perhaps an even greater advantage is that class loading is not done during run time. This means that a MIDlet won't experience slow-down when a new class is accessed. Further the MIDlet won't have to share the heap with that have been loaded from the JAR file.

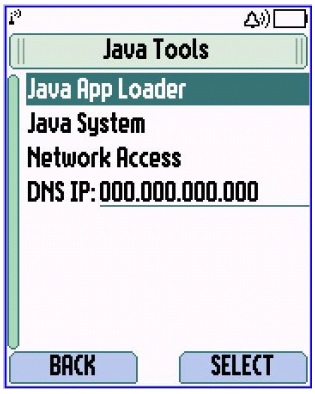

Figure 4. Java Application Loader on Java Tools

MIDlet Suite De-installation

An installed MIDlet can only be removed from the device by de-installing it from the Java Apps menu. De-installing a MIDlet Suite will remove the installed image from Program Space. The resources are then removed from Data Space along with the JAD file.

From the Games & Apps menu, the user can highlight a MIDlet Suite, selecting the menu soft key, and bring up the Delete dialog for that MIDlet Suite (Figure 5).

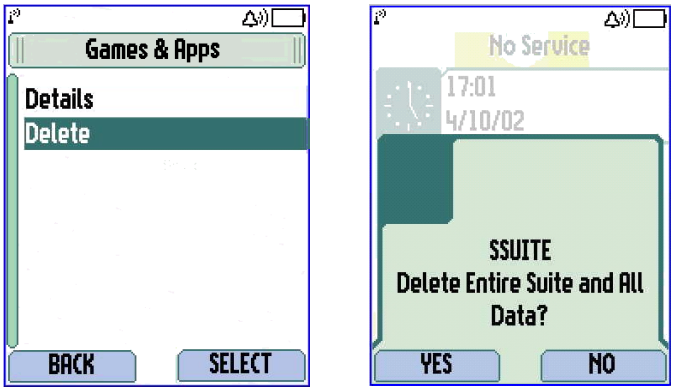

Figure 5. MIDlet Suite de-installation

MIDlet Suite Updating

When a MIDlet Suite is de-installed, all of its resources are removed including any resources that were created by the MIDlets in the suite, such as RMS databases. If a user gets a new version of a MIDlet Suite, then the user can simply download that new version to the device that has the older version installed. Once that new version is downloaded, the user will have the option to update the MIDlet Suite. This will cause the old version to be de-installed, followed by the immediate installation of the new MIDlet Suite. The only difference is that the device will prompt the user to see if resources such as RMS databases should be preserved while de-installing the old version. This prompt will only occur if such resources exist.

With such a scheme, it places the burden of compatibility on the developer. A newer version of the MIDlet Suite should know how to use, upgrade, or remove the data in the RMS databases created by older versions. This idea of forward compatibility should not be extended to backward compatibility, because the A830 device will not allow a user to update a version of a MIDlet Suite with an older or equal version of that MIDlet Suite. If the developer tries to install an older or equal version, the A830 will ignore the installation and launch the current version of the MIDlet suite.

Starting, Pausing, and Exiting

AMS Control of MIDlet State Transitions

A MIDlet has three different states: Destroyed, Active, and Paused. A MIDlet's natural state is destroyed. The AMS typically controls the transition through these states. When a user decides to launch a MIDlet, the device puts up a screen indicating that the MIDlet is transitioning through these states. The AMS controls the MIDlets through those states by calling the MIDlet's methods, `startApp()`, `pauseApp()`, and `destroyApp()`.

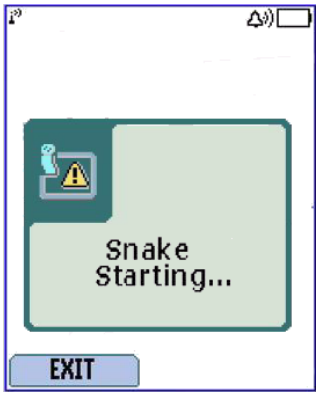

Figure 6. MIDlet Starting Screen

The constructor of the MIDlet's class that extends MIDlet is first invoked. Then its `startApp()` method is called to indicate that it's being started. The MIDlet will have focus when its `startApp()` method finishes execution. If a MIDlet takes too long initializing state variables and preparing to be run in its constructor or `startApp()` methods, it may appear to be stalled to users.

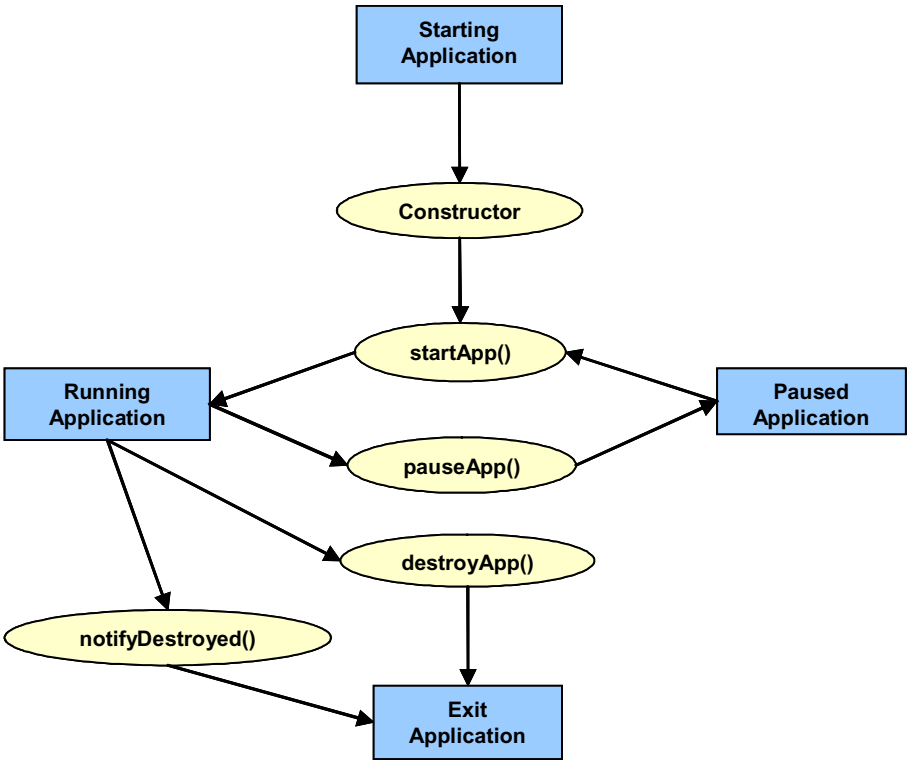

Figure 7. MIDlet State Transitions

Table 1 - State Transition Methods

Method	Caller	Purpose
Constructor	AMS	Initializes the MIDlet – should return quickly
`startApp()`	AMS	The `startApp()` method is called to start the application either from a newly constructed state or from a paused state. If the `startApp()` is called from a paused state, the MIDlet should not re-initialize the instance variables(unless it is the desired behavior). The `startApp()` method may be called multiple times during the lifespan of the MIDlet. The MIDlet may set the current display to its own Displayable from the `startApp()` method, but is shown only after the `startApp()` returns. When exiting a paused application, the KVM calls `startApp()` first followed by a call to `destroyApp()`
`pauseApp()`	AMS, MIDlet	The `pauseApp()` method is called from either AMS or from within the MIDlet.

Method	Caller	Purpose
		AMS or from within the MIDlet. The `pauseApp()` should pause active threads, and prepare for `startApp()` to be called. If the application is to be resumed with a screen other than the present, then the Displayable should be set current in the `pauseApp()`.
`destroyApp()`	AMS	The `destroyApp()` method is called from AMS and signals the MIDlet to clean up any resources to prepare for termination. For example, open RMS records should be closed, threads should be stopped, and any other housekeeping chores should be performed. The MIDlet should not call `destroyApp()`.
`notifyDestroyed()`	MIDlet	The `notifyDestroyed()` method is called by the MIDlet to exit and terminate itself. All housekeeping such as stopping active threads and closing RMS records should be performed before calling `notifyDestroyed()`. `notifyDestroyed()` notifies AMS to terminate the calling MIDlet.

Focus is an important concept. On a device without a windowing system, only one application can have focus at a time. When an application has focus, it receives keypad input, and has access to the display, speakers, LED lights, vibrator, etc. The A830 device can only run one MIDlet at a time, but that MIDlet has to share focus with the system user interface. That user interface is a higher priority than the MIDlet, so the MIDlet will immediately lose focus when the system needs to handle a phone call or some other interrupt.

The concept of focus correlates directly with the MIDlet state. i.e. when a MIDlet loses focus because of a phone call, the MIDlet is immediately paused. Conversely to the example of starting the MIDlet, the MIDlet loses focus immediately, then its `pauseApp()` method is called.

The paused state is not clearly defined by MIDP. The only requirement placed on the device manufacturer is that a paused MIDlet must be able to respond to network events and timer events. On Motorola devices, the paused state simply implies that the MIDlet is in the background as mentioned above, but it doesn't force any of the threads to stop execution. Essentially, a paused MIDlet is a MIDlet without focus whose `pauseApp()` method has been called. It's up to the developer to control their threads, such as making them sleep for longer periods, completely pausing game threads, or terminating threads that can be restarted when the MIDlet is made active again.

Similarly to the example of losing focus immediately before the `pauseApp()` method is called, a MIDlet's focus is also lost immediately before its `destroyApp()` method is

called. It's interesting to note how a Motorola device manages the transition to the destroyed state.

As described above, it is the MIDlet writer's responsibility to properly implement all methods in the javax.microedition.midlet package, especially `startApp()` and `pauseApp()`. A common error is to implement `startApp()` to execute instructions that are only intended to be executed once during MIDlet execution. The correct implementation is to include in `startApp()` those instructions which can and should be executed each time the MIDlet changes from the Paused state to the Active state. The same logic should be applied to `pauseApp()`.

The sample MIDlet below demonstrates one way of using `startApp()`. `startApp()` performs operations for the initial launch of the MIDlet as well as any operations that need to take place each time the MIDlet changes state from Paused to Active. Booleans are used to determine whether the MIDlet has started and whether it's in the Active state. These Booleans can also be used by other MIDlet threads to determine state.

```
package midp.demo;
import javax.microedition.midlet.MIDlet;

   public class Demo extends MIDlet {
   // The MIDlet has been started already
   private boolean isStarted  = false;

   // The MIDlet is in active state
   public  boolean isActive   = false;
   // (in most cases these booleans are used by other
   threads)

   protected void destroyApp(boolean unconditional){
      isActive  = false;
   }
   protected void pauseApp(){
      isActive = false;
   }
   protected void startApp(){
      isActive = true;
      if (!isStarted){
         //...Use getDisplay(), setCurrent(),
         // and other initial actions
         isStarted = true;
      }
   }
}
```

MIDlet developers should be aware that not all MIDlets found on the World Wide Web or elsewhere will necessarily execute flawlessly on all J2ME devices. This is certainly true for MIDlet state transitioning. The MIDP specification of `javax.microedition.midlet` allows for some latitude in the implementation. Therefore, it cannot be assumed that all MIDlets are perfectly compatible with all devices.

Also, some MIDlets may execute flawlessly on desktop simulators such as Sun's Wireless Toolkit [see item 4]. However, these simulators in general have no way of loosing and gaining focus such that the MIDlet transitions between the Paused and Active states and `startApp()` and `pauseApp()` are called.

MIDlet Control of MIDlet State Transitions

A MIDlet has a lot of flexibility to control its own state. A MIDlet can call its own `startApp(), pauseApp(),` and `destroyApp()` methods. However those are the methods that the AMS uses to indicate a state transition to the MIDlet, so this won't actually cause the state transition. The MIDlet can simply call those methods if it wishes to perform the work that it would typically do during that state transition.

There are another set of methods that the MIDlet can use to cause state transitions. They are `resumeRequest(), notifyPaused(),` and `notifyDestroyed().` Since the system user interface has priority, a MIDlet cannot force itself into the active state, but it can request that it be resumed via a `resumeRequest().` If the system is not busy, then it will automatically grant the request. However if the device wasn't in the idle screen, then it displays an alert dialog to ask the user if they'd like to resume the MIDlet. If the user denies the request, then the MIDlet is not notified. However if the user grants the request, the MIDlet's `startApp()` method is called, and it gains focus when that finishes.

The MIDlet does have more control when it decides that it wants to be paused or destroyed. It simply performs the necessary work by calling its own `pauseApp()` or `destroyApp()` method, then it notify the AMS of its intentions by calling `notifyPaused()` and `notifyDestroyed()` appropriately. Once notified, the AMS transition the MIDlet's state and revoke focus.

Java System

Besides managing MIDlet Suites from the Java Tools Menu, you can also perform system maintenance. The Java System feature gives statistics about the system such as:

- CLDC Version
- MIDP Version
- Data Space (Free space)
- Program Space (Free space)
- Total Heap Size

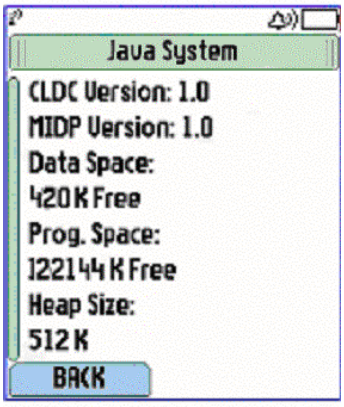

Figure 8. Java System Menu

5 Limited Connected Device User Interface (LCDUI)

Overview

The default user interface package for MIDP is LCDUI. It provides several UI components that a developer can use to build an application quickly. If more control over the UI is needed, the LCDUI Canvas class can be used to draw images, basic primitive shapes, and receive raw key presses.

The Motorola A830 handset currently supports the PNG with Transparency image type.

Class Description

The API for the LCDUI is located in the package:

`javax.microedition.lcdui`

Available Fonts

Overview

As MIDP states the **`Font`** class represents fonts and font metrics. **`Fonts`** cannot be created by applications. Instead, applications query for fonts based on font attributes and the system will attempt to provide a font that matches the requested attributes as closely as possible. A **`Font`**'s attributes are style, size, and face. The style value may be combined using the OR operator whereas the size and face attributes cannot be combined.

Fonts

The Motorola A830 handset offers different sizes, styles, and faces. The following table describes these fonts:

Font Faces	
FACE_PROPORTIONAL	Each font has a variable width and a fixed height.
FACE_MONOSPACE	Each font has a fixed width and height.
FACE_SYSTEM	Each font has a variable width and a fixed height. These fonts are used in the ergonomics of the Motorola A830 handset.

Font Sizes	
SIZE_SMALL	The Motorola A830 handset offers small size fonts for all 3 font faces.
SIZE_MEDIUM	The Motorola A830 handset offers medium size fonts for all 3 font faces.
SIZE_LARGE	The Motorola A830 handset offers large size fonts for all 3 font faces.

Font Styles	
STYLE_PLAIN	The Motorola A830 handset offers plain style fonts for all 3 font faces and sizes.
STYLE_UNDERLINED	The Motorola A830 handset offers underlined style fonts for all 3 font faces and sizes.
STYLE_BOLD	The Motorola A830 handset offers bold style fonts for all 3 font faces and sizes.
STYLE_ITALIC	The Motorola A830 handset does not offer any italic font styles.

Default Fonts

The default font is set to FACE_SYSTEM, SIZE_MEDIUM, and STYLE_PLAIN for all LCDUI components.

kJava Telephony

This feature allows the user to have ability to press SEND key and make a call using phone number from current selected **TextField** with PHONENUMBER attribute (see

Sun™ MIDP Specification [2], `javax.microedition.lcdui.TextField` class). The user is asked to confirm the action before any voice call is made. This feature asks the user to confirm the return to the application after completion of the call.

Functionality

The MIDlet application can specify a special attribute for **`TextField`** of MIDP to indicate it is a phone number. The application shows the **`TextField`** on the screen and the user should select this field before making a call.

The user presses the send key to set up a call from the **`TextField`** and the product shows a confirmation dialog and asks the user permission to setup a voice call to the number indicated in the **`TextField.`**

After user's confirmation, the Calling Application API (implemented on KVM) provides a voice call on the A830 device.

After call termination, the A830 shows a confirmation dialog and asks the user to abort or to return to the MIDlet application execution.

Selecting the return to the application, the execution of the KVM is suspended to guarantee return to current state of the application.

The current implementation of **`TextField`** supports only digits in a field with PHONENUMBER attribute. The following characters can be added to a phone number digit string and are stored in Motorola A830 handsetbook:

- Pause character to create a timed delay during call setup. It is represented by a lower case 'p'.
- Wait character to create an untimed delay during call setup. It is represented by a lower case 'w'.
- A 'n' character is used to represent a variable phone number to be selected during call setup.
- An international dialing prefix for GSM, it is represented as '+' character.

Code Examples

Below is a simple example on how to create a simple traversable list using standard LCDUI widgets. In this example, we create a simple menu that allows the selection of multiple food items.

```
/* Our list of foods */

String foodList[] = {“Apple”, ”Cookies”, ”Cake”, ”Oranges”,
”Cheese Burger”, ”Ice Cream”};

/* create a form with a title List Form */

Form myForm = new Form(“Food Menu”);
```

```
/**
 * Create a choice group with the foodList.
 * Also make the choice group be able to accept multiple
 * Choices from the list.
 */
ChoiceGroup foodChoice = new
  ChoiceGroup(“Lunch”,Choice.MULTIPLE,foodList,null);

/* append the list to our form */
myForm.append(foodChoice);

/* now display the menu */
display.setCurrent(myForm);
```

Tips

Although the Canvas class provides a high level of control over a display, using a canvas for every screen produces larger classes and more of them. Developers can save a lot of space in the MIDlet Suites, if they make use of standard LCDUI components when possible.

Caveats

When using Canvas's `getWidth()` and `getHeight()` methods, the available screen area is returned. The available screen area is the full screen area excluding the command soft key area. If your Canvas possesses LCDUI Command objects, that area will be used uniquely for rendering of the commands. This will cause the total available space for drawing to be reduced by the amount of space the commands take up. The amount of space required for the command area is equal to the font height plus two pixels for borders.

6 Lightweight Window Toolkit (LWT)

Overview

The Lightweight Window Toolkit (LWT) is an extension of the Java 2 Platform, Micro Edition (J2ME) MIDP specification. LWT addresses the limitations of the MIDP user interface APIs known as LCDUI. Specifically, LCDUI does not provide a developer with complete control over screen layouts, nor does it permit using custom components or extending existing components. LWT solves these problems. Designed to enhance user interface capabilities, LWT is a key enabler for the development of full-featured applications on mobile devices. It is especially valuable for delivering a rich user experience on more capable mobile devices otherwise constrained by LCDUI's limited capabilities.

The Motorola LWT API is described in more details on LWT Programmer's Guide .

Example of a MIDlet using LWT package

The following example illustrates the creation of the classical "Hello World" program using the LWT package.

```
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;
import com.motorola.lwt.*;
import java.io.IOException;

public class HelloLWTWorld extends MIDlet {
   Display display;
   ComponentScreen scr;
   ImageLabel label;
```

```
    public HelloLWTWorld() {
        // get display
        display = Display.getDisplay(this);

        // create the ComponentScreen
        scr = new ComponentScreen();

        // create ImageLabel
        label = new ImageLabel(null, null, "Hello LWT
        World!");

        // place the ImageLabel in the center of the screen
        label.setLeftEdge(Component.SCREEN_HCENTER,
                                  -
        label.getPreferredWidth()/2);
        label.setTopEdge(Component.SCREEN_TOP,
                         (scr.getHeight()-
        label.getPreferredHeight())/2);

        // add ImageLabel to the ComponentScreen
        scr.add(label);
    }

    public void startApp() {
        display.setCurrent(scr);
    }

    public void pauseApp() {

    }

    public void destroyApp(boolean b) {

    }
}
```

The main concept illustrated in the example is the creation of a container, the ComponentScreen, and the addition of a Component, the ImageLabel. This example will be revisited later to cover the details regarding the location and dimension of the Component.

Class hierarchy and Overview

The following diagram shows the class hierarchy of LWT.

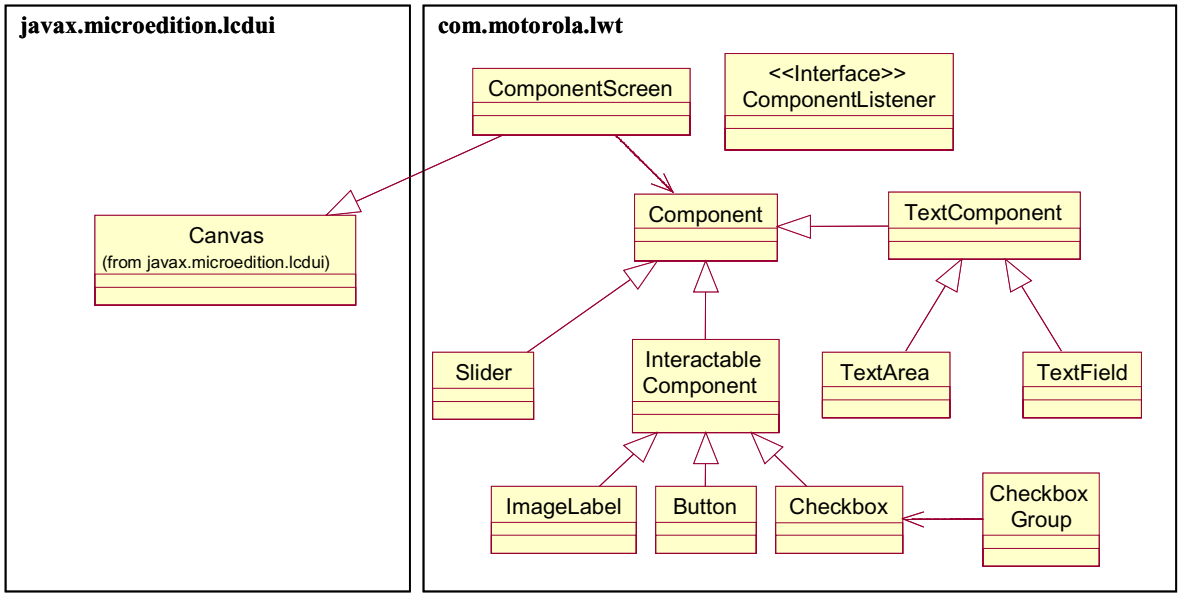

Figure 9. LWT class hierarchy

ComponentScreen

The **`ComponentScreen`** class is the top-level container in an LWT user interface. As a subclass of LCDUI's **`Canvas`**, it can be interchanged with other LCDUI screens such as **`Canvas, Form`**, and **`Alert`**.

`ComponentScreen` inherits several methods from **`Canvas`** that provide the mechanisms for handling input events and repainting; thus, the interface to LCDUI is accomplished using the published APIs, and LWT can be integrated with any MIDP-compliant implementation.

Component

`Component` is the abstract base class of all LWT user interface entities that can be added to a **`ComponentScreen`**.

ComponentListener

The **ComponentListener** interface is implemented by any class that receives events from a **Component**. The **ComponentListener** is notified of an event by calling its processComponentEvent method with the source **Component** reference and an integer identifying the event type.

InteractableComponent

InteractableComponent is the abstract base class of the components that a user can 'press' and 'release'. Such components include buttons, checkboxes, and icons. This class serves to reduce code size and complexity of its subclasses by providing the basic interaction functionality. An **InteractableComponent** is actuated by tapping and releasing within its bounds, or by pressing and releasing the Enter key when the component has focus.

Button

Button is a basic button that a user can actuate. A button can display text to convey its meaning. The text font is the only customizable attribute of the **Button** class.

ImageLabel

An **ImageLabel** is a general-purpose component that can display an image and/or a text label; it can be an interactive or a read-only component.

Image

If an image is displayed, a developer can use either a single image or multiple images to reflect a component's different states (such as pressed, disabled, etc.).

Text

If text is displayed, a developer can specify the text, its color, and its font.

Label Location

If both text and an image are displayed, the location of the text relative to the image can be specified as:

- Above: The label is displayed centered above the image
- Below: The label is displayed centered below the image

- Left: The label is displayed centered to the left of the image
- Right: The label is displayed centered to the right of the image
- Centered: The label is displayed centered on the image

Alignment

Regardless of what is displayed (text, image, or both), the collective alignment of the image and/or text within the bounds of the **`ImageLabel`** may be specified as:

- North: The image and label both are displayed in the top half of the **`ImageLabel`**
- South: The image and label both are displayed in the bottom half of the **`ImageLabel`**
- East: The image and label both are displayed in the right half of the **`ImageLabel`**
- West: The image and label both are displayed in the left half of the **`Imagelabel`**
- Centered: The image and label both are displayed centered on the **`ImageLabel`**

Checkbox

Used as is, **`Checkbox`** provides a single, independent boolean choice. A **`Checkbox`** displays text to convey its meaning. For example, a series of **`Checkboxes`** can allow a user to select toppings for a burger.

CheckboxGroup

A **`CheckboxGroup`** is a non-UI object that manages one or more **`Checkboxes`**. It can be configured to enforce multiple selection or exclusive selection rules, and can be used to query the current values of the checkboxes. When used in conjunction with a **`CheckboxGroup`**, **`Checkboxes`** can provide a list of exclusive choices in the form of radio buttons or a list. To continue the above example, a user can select how a burger is cooked using several **`Checkboxes`** and a **`CheckboxGroup`**, either as a series of radio buttons or a list. **`Checkboxes`** can be added to or removed from a **`CheckboxGroup`** as needed. **`Checkboxes`** must still be added to the **`ComponentScreen`**; adding them to the **`CheckboxGroup`** only impacts their behavior.

TextComponent

`TextComponent` is the abstract base class for components that can display and edit text. It provides common functionality such as text manipulation, constraints, and input event handling.

TextField

`TextField` is a single-line **`TextComponent`** designed to display and edit text. **`TextField`** supports horizontal scrolling only.

TextArea

`TextArea` is a multi-line **`TextComponent`** designed for displaying and editing text. **`TextArea`** supports vertical scrolling only.

Slider

The **`Slider`** is a gauge-type component that provides a graphical representation of a numeric value. A **`Slider`** can be read-only or adjustable. A read-only **`Slider`** might be used to indicate memory usage or battery level; an adjustable **`Slider`** might be used to adjust a volume level. The current value of a **`Slider`** represents the current setting or level of the **`Slider`**, which can be between 0 and the *maximum value*, inclusive. The *maximum value* of a **`Slider`** may be set programmatically to any non-negative integer value. A **`Slider`** does not include a label. A developer can add labels and icons to the **`ComponentScreen`** to indicate meanings, endpoint values, etc.

Fundamental Component Behaviors

At the heart of LWT are the **`ComponentScreen`** and **`Component`** classes; together, they provide the bulk of LWT's basic functionality. This section describes the fundamental behaviors exhibited by **`ComponentScreen`** and **`Component`**. There are compelling reasons for describing these behaviors and their mechanisms in detail. First, it allows developers to fully exploit the APIs and minimize redundant code. Second, it enables developers to customize behavior without the risk of side effects.

Component Management

Containership Rules

`Components` can be added and removed from a **`ComponentScreen`**. A **`ComponentScreen`** cannot be added to another **`ComponentScreen`**, and a **`Component`** cannot be added to another **`Component`**. A **`Component`** can have only one parent **`ComponentScreen`** at a time, and it can be added to a given **`ComponentScreen`** only once. Whenever a **`Component`** is added to a

ComponentScreen, it is first removed from the current parent if one exists, thereby ensuring that these two rules are enforced.

Component Indices

A **ComponentScreen** maintains an ordered list of its child components and assigns each one a unique index. The index of a component indicates its position in the list where 0 is the first component, and the highest index is the last component. Indices are always consecutive, so the index for a given component may change if other components are added or removed from the same **ComponentScreen**. For example, if a component is added in the middle of the list, the indices of the subsequent components will be incremented to account for the inserted component.

A component may be inserted at a specific valid index, or it may be simply appended at the end of the list and automatically assigned the next index. A component's index is significant since it implies Z-order and dictates the order in which layout and focus traversal are performed.

Z-Order

The component with the highest index is considered closest to the user, as shown below.

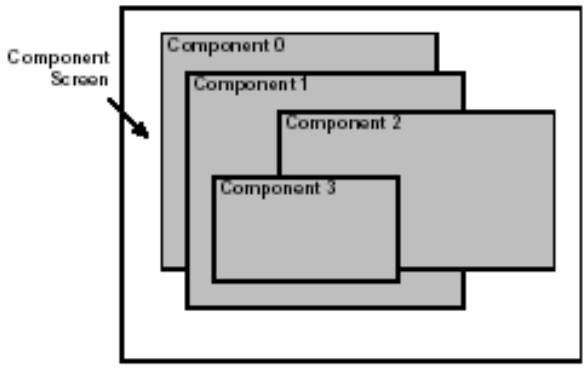

Figure 10. Z-Order

Component Regions

Each component occupies a rectangular region of its parent **ComponentScreen**. A component receives pointer events that occur within its rectangular region, and is responsible for rendering the pixels within its region. A component may render itself as an ellipse, a triangle, a cloud, etc., but its bounding region is always rectangular.

Region Parameters

The region is fully described by the location of the upper-left corner of the component and by the component's width and height. The location of the upper-left corner is relative to the ComponentScreen's origin and is based on the MIDP coordinate system. Width and height are expressed in terms of pixels. A developer can query the bounds of a component by calling `getX()`, `getY()`, `getWidth()`, and `getHeight()` on the component.

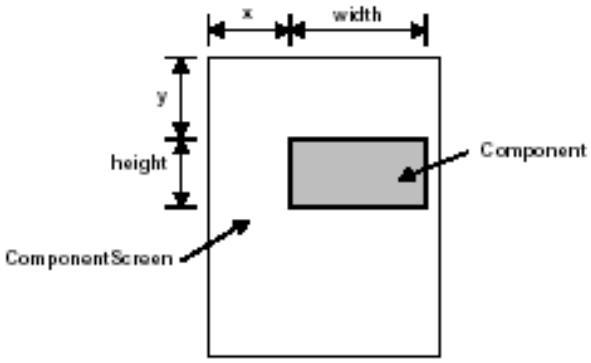

Figure 11. Region Parameters

Preferred Size

Each **`Component`** subclass must implement the methods `getPreferredWidth()` and `getPreferredHeight()`. Together, these two methods specify the ideal dimensions of a given component instance. Even for the same class, different instances may specify different preferred sizes to reflect the length of a text label, size of an image, etc. The `preferredWidthChanged()` method must be called whenever the preferred width of the component changes. Similarly, the `preferredHeightChanged()` method must be called whenever the preferred height of the component changes. For standard LWT components, these methods are automatically called when a relevant parameter is changed; for custom components, it is a developer's responsibility to call these methods whenever a change is made that impacts the preferred width or height of the component.

Component States

Each **`Component`** instance carries state information. Subclasses may introduce additional state information as needed. The pre-defined states are:

- Visibility
- Enabling

Visibility

A visible component is shown to a user, whereas an invisible component is not. Components may be hidden to conceal portions of the user interface that are not relevant, thereby simplifying the user interface. By default, all components are initially visible.

Enabling

Enabling indicates whether or not a component is currently available to a user. Enabling and disabling is useful for conveying the availability of certain features that may be temporarily unavailable based on the current context. For example, a View button should be disabled if the corresponding list contains no items. By default, all components are initially enabled.

Component Layout

To meet LWT's design goals, the layout model is designed to provide a developer with complete control over component placement and size. Although this approach provides the greatest flexibility, it can result in fairly large applications, especially if the application must automatically adjust its layout to account for different display and component sizes. Therefore, the LWT layout model also incorporates several features that enable the creation of adaptable complex layouts with very little code; furthermore, the execution of these layouts is inherently efficient.

Layout Model

A component's region is specified in terms of its left, right, top, and bottom edges.

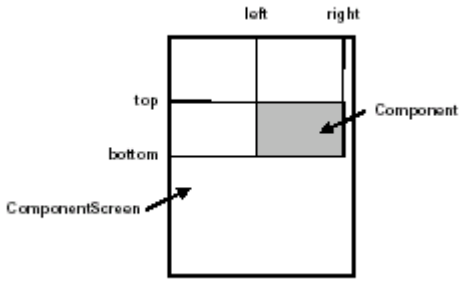

Figure 12. Component Layout

A developer can independently specify the location of each edge using one of several schemes. An accompanying value controls the location of the edge according to the scheme selected.

Offset Conventions

For all schemes that use an offset, the offset values extend down and to the right. That is, a horizontal offset extends to the right for positive values and to the left for negative values. Similarly, a vertical offset extends down for positive values and up for negative values.

Centering

Whenever components are centered, the location is obtained by truncating the mean of the two endpoints. This convention permits the use of a bit shift rather than a more complex division operation to determine the center. Mathematically, the center point between A and B is defined as (A + B) >> 1.

Left Edge

The following schemes may be used for specifying the location of a component's left edge:

Scheme	Behavior
`SCREEN_LEFT (default)`	The accompanying value describes the edge's offset from the left edge of the screen.
`SCREEN_HCENTER`	The accompanying value describes the edge's offset from the center of the screen.
`SCREEN_RIGHT`	The accompanying value describes the edge's offset from the right edge of the screen.
`PREVIOUS_COMPONENT_LEFT *`	The accompanying value describes the edge's offset from the left edge of the previous component.
`PREVIOUS_COMPONENT_HCENTER *`	The accompanying value describes the edge's offset from the center of the previous component;
`PREVIOUS_COMPONENT_RIGHT *`	The accompanying value describes the edge's offset from the right edge of the previous component.

* Interpreted as `SCREEN_LEFT` if there is no previous component.

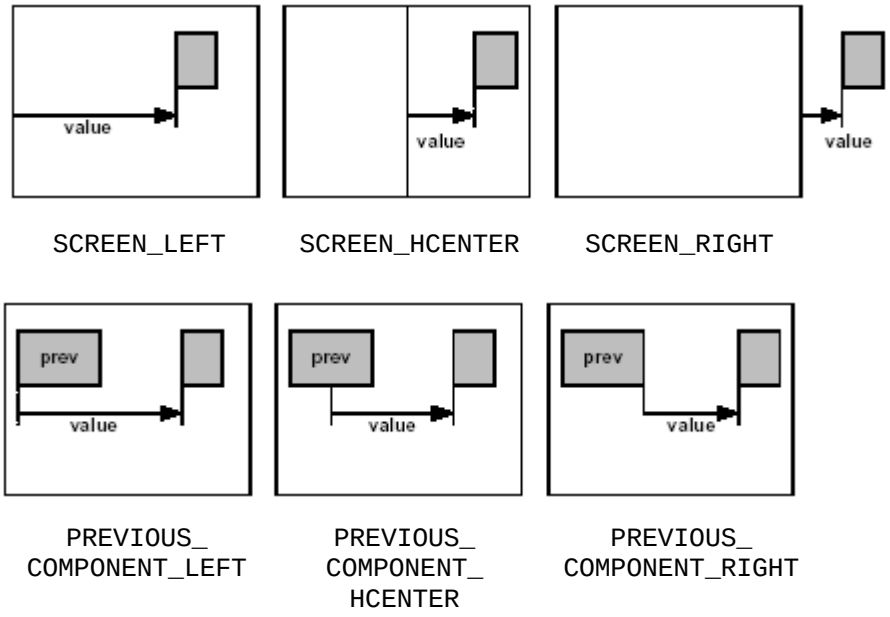

Figure 13. Left edge

Right Edge

The following schemes may be used for specifying the location of a component's right edge. Developers should use `PREFERRED_WIDTH` wherever feasible to maximize application portability across different devices.

Scheme	Behavior
`SCREEN_LEFT`	The accompanying value describes the edge's offset from the left edge of the screen
`SCREEN_HCENTER`	The accompanying value describes the edge's offset from the center of the screen
`SCREEN_RIGHT`	The accompanying value describes the edge's offset from the right edge of the screen
`PREVIOUS_COMPONENT_LEFT *`	The accompanying value describes the edge's offset from the left edge of the previous component
`PREVIOUS_COMPONENT_HCENTER *`	The accompanying value describes the edge's offset from the center of the previous component
`PREVIOUS_COMPONENT_RIGHT *`	The accompanying value describes the edge's offset from the right edge of the previous component

Scheme	Behavior
WIDTH	The right edge is located such that the component's width is equal the accompanying value
PREFERRED_WIDTH (default)	The right edge is located such that the component's width is equal to its preferred width plus the accompanying value

* The component is set to its preferred width if there is no previous component.

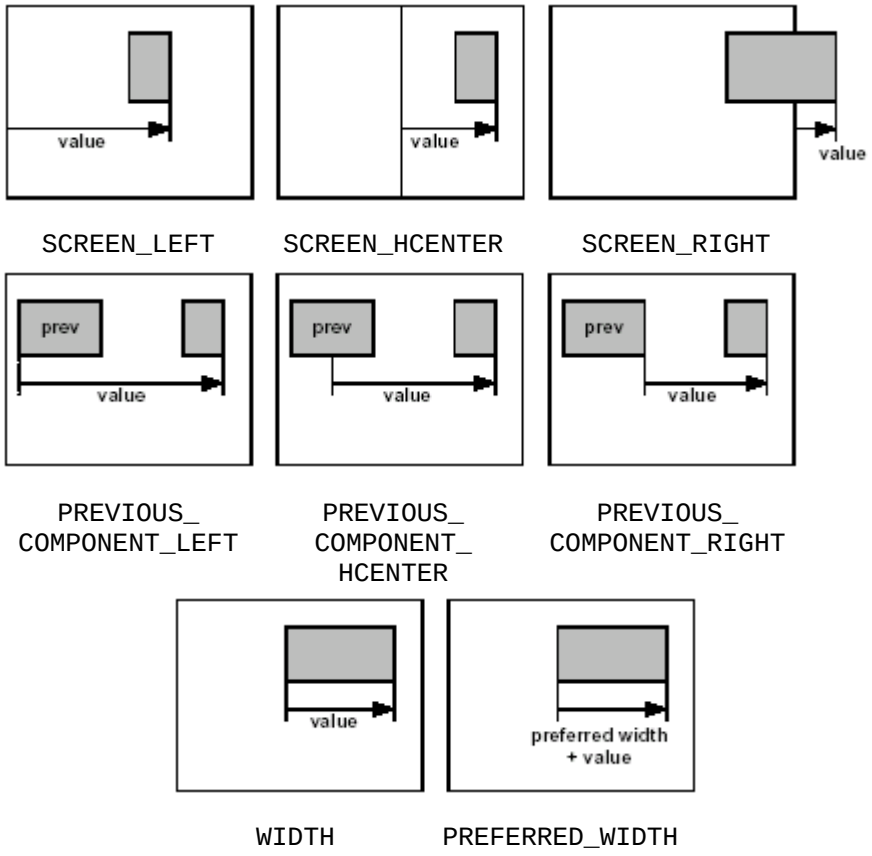

Figure 14. Right Edge

Top Edge

The following schemes may be used for specifying the location of a component's top edge:

Scheme	Behavior

SCREEN_TOP	The accompanying value describes the edge's offset from the top edge of the screen.
PREVIOUS_COMPONENT_TOP *	The accompanying value describes the edge's offset from the top edge of the previous component.
PREVIOUS_COMPONENT_VCENTER *	The accompanying value describes the edge's offset from the center of the previous component.
PREVIOUS_COMPONENT_BOTTOM *(default)	The accompanying value describes the edge's offset from the bottom edge of the previous component.

* Interpreted as SCREEN_TOP if there is no previous component.

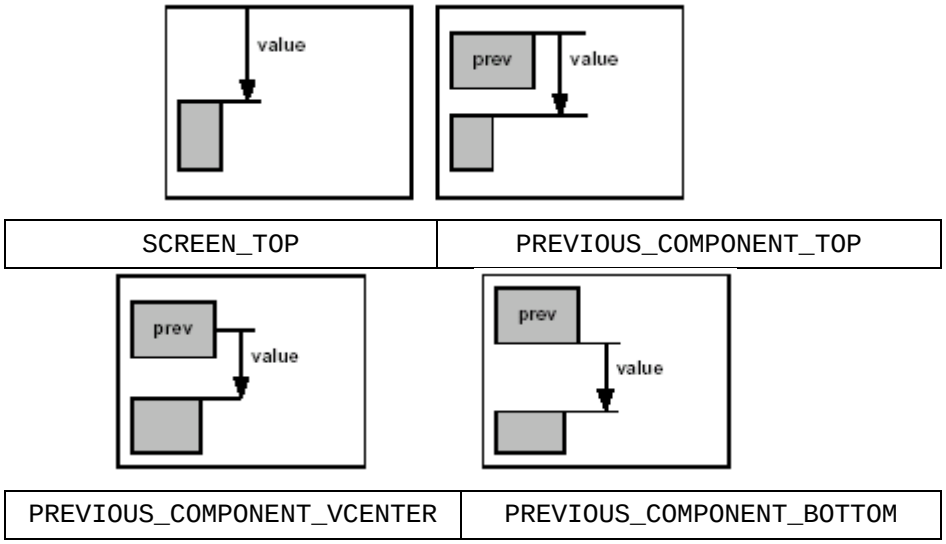

Figure 15. Top Edge

Bottom Edge

The following schemes may be used for specifying the location of a component's bottom edge. Developers should use PREFERRED_HEIGHT wherever feasible to maximize application portability across different devices.

Scheme	Behavior
SCREEN_TOP	The accompanying value describes the edge's offset from the top edge of the screen
PREVIOUS_COMPONENT_TOP *	The accompanying value describes the edge's offset from the top edge of the

Scheme	Behavior
	previous component
`PREVIOUS_COMPONENT_VCENTER *`	The accompanying value describes the edge's offset from the center of the previous component
`PREVIOUS_COMPONENT_BOTTOM *`	The accompanying value describes the edge's offset from the bottom edge of the previous component
`HEIGHT`	The bottom edge is located such that the component's height is equal the accompanying value
`PREFERRED_HEIGHT (default)`	The bottom edge is located such that the component's height is equal to its preferred height plus the accompanying value

* The component is set to its preferred height if there is no previous component.

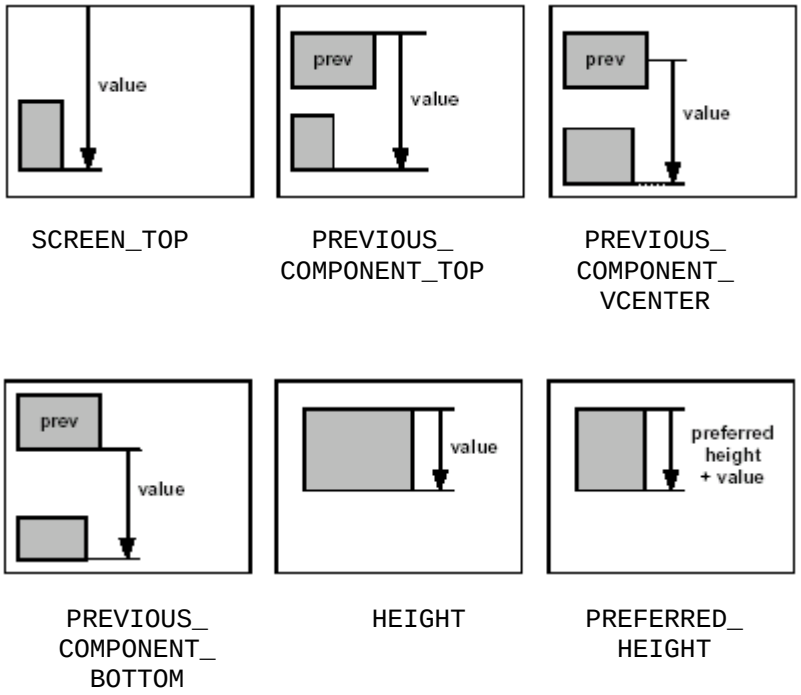

Figure 16. Bottom Edge

Validation Cycle

To minimize redundant layout computations, the **`ComponentScreen`** tracks the state of its layout and computes its layout only when necessary. This mechanism

effectively consolidates requests for layout computation and defers the layout process until updated component bounds are actually needed.

Invalidation

A **`ComponentScreen`** becomes *invalid* when a change is made that could potentially alter the layout of its components. Such changes include adding or removing components and changing the edge specifications or visibility of a child component. When such a change is made, the affected **`ComponentScreen`** automatically becomes invalid. A **`ComponentScreen`** can be programmatically made invalid by calling `invalidate()`.

Changes to Preferred Width and Height

For some components, the preferred size is dependent on component-specific attributes such as label width, image size, font, etc. In such cases, a change to one of these attributes may result in a change to the preferred width or height. When such a change occurs, the component must call `preferredWidthChanged()` or `preferredHeightChanged()`, respectively. These methods invalidate the parent if the preferred dimension is currently being used for the component's layout; otherwise the change is irrelevant and invalidation is not required.

Validation Process

An invalid **`ComponentScreen`** becomes *valid* by ensuring that the layout of its children is up to date. The process of validation involves checking whether or not the **`ComponentScreen`** is invalid; if so, the `doLayout()` method is called and the **`ComponentScreen`** then becomes valid.

Layout Process

The `doLayout()` method computes the location of left, right, top, and bottom edges for each component based on their schemes and accompanying values. The components are processed in ascending index order. Invisible components are ignored by the layout process; their edges are not computed and they never become the previous component.

Validation Triggers

Validation automatically occurs prior to any operation that relies on accurate component layout information, specifically rendering and pointer event dispatching. A developer can programmatically force validation to occur by calling `validate()`.

Focus Management

Each **ComponentScreen** instance keeps track of its current *focus owner*. The *focus owner* is the component within the **ComponentScreen** that receives key events. By default, the focus owner is null indicating that no component is currently receiving key events; in this case, the **ComponentScreen** continues to receive key events but does not dispatch them to a component. The focus owner may also become null if the current focus owner is removed or is no longer eligible to maintain focus.

Focus Acceptance

A component may indicate whether or not it is interested in ever becoming the focus owner by setting the Boolean field acceptsKeyFocus to the appropriate value. If this field is set to true, the component may gain focus; if false, the component will never gain focus.

Focus Eligibility

In order to be eligible to gain focus, the component must be visible, enabled, and have its acceptsKeyFocus field set to true.

Focus Traversal

The user may traverse focus by the appropriate keys on the device. Focus traversal occurs in component index order and skips any components that are not eligible to receive focus. Focus traversal wraps from the last component to the first component and vice versa. Focus traversal can also be triggered programmatically by calling `focusNext(), focusPrevious(), focusFirst(),` and `focusLast()`.

Requesting Focus

A component may programmatically request focus by calling `requestFocus()`.

Focus Notifications

Whenever a component gains key focus, its `gainedFocus` method is called. Similarly, its `lostFocus` method is called whenever it loses focus. A component can query whether or not it has focus by calling `hasFocus()`.

Key Event Handling

Key events are dispatched to the current **ComponentScreen** through the three methods defined in LCDUI's **Canvas**. The default implementations of these methods in **ComponentScreen** check if there is a current focus owner and dispatch the event

to that component, if any. Subclasses may override these methods to implement custom key event handling.

Key events are dispatched to the component through these three methods: `keyPressed`, `keyRepeated`, and `keyReleased`. These methods return a Boolean to indicate if the component consumed the key event, thereby allowing **`ComponentScreen`** subclasses to implement default behaviors for unconsumed key events.

Pointer Event Handling

Pointer events are dispatched to the current **`ComponentScreen`** by means of the three methods defined in LCDUI's **`Canvas`**. The default implementations of these methods in **`ComponentScreen`** dispatch these events to the appropriate *target* component. Subclasses may override these methods to implement custom pointer event handling.

Pointer Event Targeting

A component becomes the target when a pointer-pressed event occurs within its bounds. The search for the component is performed in descending index order to implement the correct Zorder; in the event that components overlap, the component with the highest index (i.e., closest to the user) becomes the target. Invisible and disabled components are ignored when searching for the target.

Once it becomes the target, a component continues to be the target until the pointer is released. Therefore, a component may receive pointer-drag and release events outside of its bounds.

Rendering

Component Screen Rendering

The **`ComponentScreen`** is rendered by a call to its `paint()` method. By default, this method first clears the background (i.e., fills it with white pixels) and then renders its components by calling `paintComponents`. **`ComponentScreen`** subclasses may override the default `paint()` method to implement special backgrounds or to render other artifacts on the screen.

The `paintComponents` method renders the components in ascending index order. If the components overlap, the component with the highest index is rendered last and appears to be closest to the user, thereby implementing the correct Z-order. Invisible components are not rendered.

Component Rendering

A component is rendered by a call to its `paint` method. The provided Graphics object is translated such that its origin is located at the upper-left corner of the component. Also, the clip region of the Graphics object is intersected with the bounds of the component.

Components Are Transparent

Since the **`ComponentScreen`** is responsible for rendering the background, **`Component`** does not clear the background prior to rendering. Rendering of the background by **`Component`** is redundant and reduces performance.

States That Impact Appearance

A component must render itself in a manner that conveys its current state to the user. All components must render themselves to reflect the following mutually exclusive states:

- **Normal –** Normal appearance
- **Disabled –** Should be grayed out or drawn with dotted lines instead of solid lines.
- **Focus Owner –** Normal appearance with a thick border (a component needs to support this state only if it accepts key focus)

Component subclasses may include additional states or attributes that affect their appearance; these should also be accounted for by the rendering code.

Scrolling

`ComponentScreen` supports vertical scrolling, but does not support horizontal scrolling.

Enabling and Disabling Scrolling

Scrolling is automatically enabled by the native user interface if the bottom edge of the last component extends past the bottom of the screen. In other words, scrolling support is provided when it is needed, and may be removed when it is not needed.

Focus-Driven Scrolling

Whenever a component receives key event focus, the screen is automatically scrolled when necessary to ensure that the component is visible to the user.

User Interface Scrolling

It is the responsibility of the implementation and native user interface to provide the user with the ability to control the scroll position.

Programmatic Scrolling

A developer can query and set the scroll position programmatically. However, a developer is not permitted to explicitly enable or disable scrolling since that functionality is implicitly provided by the device.

The ComponentScreen Class

The **`ComponentScreen`** class extends the **`Canvas`** class, and forms the basis for all LWT screen layouts. An application defines a user interface screen by creating a **`ComponentScreen`** and then adding the desired LWT components to it. With this class, a developer can create a screen from an arbitrary mix of components, including special component subclasses. It also provides the developer with complete control over the screen layout.

ComponentScreen Definition and Constructor

The **`ComponentScreen`** class is defined by: `public class ComponentScreen extends javax.microedition.lcdui.Canvas`

Its only constructor is:

- public ComponentScreen()

ComponentScreen Methods

In addition to the methods described in this section, a **`ComponentScreen`** inherits many methods from the `javax.microedition.lcdui.Canvas`, `java.lang.Object` and `javax.microedition.lcdui.Displayable` classes. These inherited methods are listed in the LWT API documentation.

The methods specifically defined by **`ComponentScreen`** are:

- `void add (Component w)` – adds a `Component` to the `ComponentScreen;`
- `protected void doLayout ()` – recomputes the layout of the `Components` according to the edge specifications for each component;
- `Component getComponent (int index)` – gets the `Component` at the specified index;
- `int getComponentCount ()` – gets the number of `Components` currently contained in this screen;

- `Component getFocusOwner ()` – gets the `Component` that currently has key event focus;
- `int getScrollOffset()` - gets the current vertical scroll offset;
- `int getWidth()` - gets the width of the `ComponentScreen;`
- `void insert(Component comp, int index)` - inserts a `Component` to the screen at the specified index;
- `void invalidate()` - invalidates this `ComponentScreen`, indicating that its component layouts need to be recalculated;
- `protected void keyPressed(int keyCode)` - called when a key is pressed;
- `protected void keyReleased(int keyCode)` - called when a key is released;
- `protected void keyRepeated(int keyCode)` - called when a key is repeated (held down);
- `void paint(javax.microedition.lcdui.Graphics g)` - renders the screen;
- `protected void paintComponents(javax.microedition.lcdui.Graphics g)` – renders the `Components;`
- `protected void pointerDragged(int x, int y)` - called when the pointer is dragged;
- `protected void pointerPressed(int x, int y)` - called when the pointer is pressed;
- `protected void pointerReleased(int x, int y)` - called when the pointer is released;
- `void remove(Component comp)` - removes the specified `Component` from the screen;
- `void remove(int index)` - removes the `Component` at the specified index from the screen;
- `void removeAll()` - removes all `Components` from this screen;
- `void scrollTo(Component comp)` – scrolls to the specified `Component`. This method ensures that the screen's scroll position is adjusted to show as much as possible of the specified `Component;`
- `void setFocusFirst()` - moves key event focus to the first `Component` that accepts focus;
- `void setFocusLast()` - moves key event focus to the last `Component` that accepts focus;
- `void setFocusNext()` - moves key event focus to the next `Component` that accepts focus;

- `void setFocusPrevious()` - moves key event focus to the previous `Component` that accepts focus;
- `void setScrollOffset(int offset)` - sets the vertical scroll offset;
- `protected void showNotify()` - called when the `ComponentScreen` is shown;
- `void validate()` - validates this `ComponentScreen;`

Detailed information about using these methods is available in the LWT API documentation [3].

The first LWT class you should instantiate is a **`ComponentScreen`**, which will form the basis for laying out the LWT components to be displayed. If your application will use several screens, it may be worth creating a subclass that all of the screens can inherit from. For example:

```
/**
* Superclass for all of the demo screens, provides the
next/previous commands
**/

class DemoScreen extends ComponentScreen {
   public DemoScreen() {
      Command next = new Command("Next", Command.OK, 1);
      Command prev = new Command("Previous", Command.BACK,
      1);
      addCommand(next);
      addCommand(prev);
   }
}
```

This class allows you to build a number of screens which have 'Previous' and 'Next' command buttons in addition to whatever components you decide to place on the individual screens.

The Component Class

The **`Component`** class is the abstract base class from which the various LWT components are descended. Each subclass descended from the **`Component`** class must implement methods to render the **`Component`**, provide preferred width and height, and optionally, to handle events.

A **`Component`** is a single user interface object that occupies a rectangular region of its parent **`ComponentScreen`**. A **`Component`** can belong only to a single screen. If a program attempts to add a **`Component`** to a second screen, it will first be removed from its current screen (which could be the screen that it is being added to) before it is added to the second screen.

The location and size of a **`Component`** is determined by specifying where the left, right, top and bottom edges are located. The location of each edge can be specified using

one of several schemes, including offsets relative to the preceding **Component**, as defined in Section 3.4 of this document. By default, a **Component** will be set to its preferred size, and will be aligned with the left edge of the screen directly beneath the preceding **Component**.

A **Component** must provide a paint method to render itself. The **Graphics** object passed to the **Component's** `paint()` method is translated so that the origin is located at the upper left corner of the **Component**.

If desired, a **Component** can respond to key and pointer events by overriding the appropriate methods (`keyPressed`(), `pointerDragged`(), etc.) In order to receive key event focus, a **Component** must have `acceptsKeyFocus` set to `true`, and it must be visible and enabled. Provided the parent screen is shown, the **Component** with key event focus will receive all key events.

Component Definition and Constructor

The **Component** class is defined by:

```
public abstract class Component extends java.lang.Object
```

Its only constructor is:

- public Component()

Component Fields

These constants define the values used by the programmer to define where the **Component** code should place and size the **Component** on the screen, as well as to define its ability to interact with the user. These constants are typically passed to some of the methods defined below to indicate from where the **Component** location value is to be measured.

- `protected boolean acceptsKeyFocus` – indicates if this Component accepts key focus (that is, it uses key events)
- `static int HEIGHT` – the bottom edge is located such that the Component's height is equal to the accompanying value
- `static int PREFERRED_HEIGHT` – the bottom edge is located such that the Component's height is equal to its preferred height plus the accompanying value
- `static int PREFERRED_WIDTH` – the right edge is located such that the Component's width is equal to its preferred width plus the accompanying value
- `static int PREVIOUS_COMPONENT_BOTTOM` – the value describes the edge's offset from the bottom edge of the previous Component
- `static int PREVIOUS_COMPONENT_HCENTER` – the value describes the edge's offset from the horizontal center of the previous Component

- `static int PREVIOUS_COMPONENT_LEFT` – the value describes the edge's offset from the left edge of the previous Component
- `static int PREVIOUS_COMPONENT_RIGHT` – the value describes the edge's offset from the right edge of the previous Component
- `static int PREVIOUS_COMPONENT_TOP` – the value describes the edge's offset from the top edge of the previous Component
- `static int PREVIOUS_COMPONENT_VCENTER` - the value describes the edge's offset from the vertical center of the previous Component
- `static int SCREEN_HCENTER` - the value describes the edge's offset from the center of the screen
- `static int SCREEN_LEFT` - the value describes the edge's offset from the left edge of the screen
- `static int SCREEN_RIGHT` - the value describes the edge's offset from the right edge of the screen
- `static int SCREEN_TOP` - the value describes the edge's offset from the top edge of the screen
- `static int WIDTH` - the right edge is located such that the Component's width is equal to the accompanying value

Component Methods

In addition to the methods defined directly by the **`Component`** class, this class inherits several methods from the **`java.lang.Object`** class. See the LWT API documentation for details on these inherited methods. The methods defined by the **`Component`** class are:

- `boolean acceptsFocus()` - checks if this Component currently accepts key focus
- `void gainedFocus()` - called when this Component gains key focus
- `int getHeight()` - gets the height of the Component, in pixels
- `ComponentScreen getParent()` - obtains a reference to the Component's parent screen
- `abstract int getPreferredHeight()` - gets the preferred height of this Component
- `abstract int getPreferredWidth()` - gets the preferred width of this Component
- `int getWidth()` - gets the width of the Component, in pixels
- `int getX()` - gets the x coordinate of the Component's left edge within the parent
- `int getY()` - gets the y coordinate of the Component's top edge within the parent

- `boolean hasFocus()` – checks if this Component currently has key focus (that is, it is receiving key events). For a given screen, no more than one Component can have key focus
- `protected void invalidateParent()` - invalidates this Component's parent screen, if any
- `boolean isEnabled()` - checks if this Component is currently enabled (can be interacted with by the user)
- `boolean isVisible()` - checks if this Component is visible (can be seen by the user)
- `protected boolean keyPressed(int keyCode)` - called when a key is pressed
- `protected boolean keyReleased(int keyCode)` - called when a key is released
- `protected boolean keyRepeated(int keyCode)` - called when a key is repeated (held down)
- `void lostFocus()` - called when this Component loses key focus
- `abstract void paint(javax.microedition.lcdui.Graphics g)` - renders the Component
- `protected void pointerDragged(int x, int y)` - called when the pointer is dragged
- `protected void pointerPressed(int x, int y)` - called when the pointer is pressed within this Component
- `protected void pointerReleased(int x, int y)` - called when the pointer is released
- `protected void preferredHeightChanged()` - notifies the system that the preferred height of this Component has changed
- `protected void preferredWidthChanged()` - notifies the system that the preferred width of this Component has changed
- `void repaint()` - requests a repaint for the entire Component
- `void repaint(int x, int y, int width, int height)` - requests a repaint for the specified portion of this Component
- `void requestFocus()` - requests key focus for this Component
- `void setBottomEdge(int scheme, int value)` - specifies the location of the Component's bottom edge. The scheme parameter is one of the constants defined above, specifying from where the value is to be measured.
- `void setEnabled(boolean b)` - sets this Component as enabled or disabled
- `void setLeftEdge(int scheme, int value)` - specifies the location of the Component's left edge. The scheme parameter is one of the constants defined above, specifying from where the value is to be measured.

- `void setRightEdge(int scheme, int value)` - specifies the location of the Component's right edge. The scheme parameter is one of the constants defined above, specifying from where the value is to be measured.
- `void setTopEdge(int scheme, int value)` - specifies the location of the Component's top edge. The scheme parameter is one of the constants defined above, specifying from where the value is to be measured.
- `void setVisible(boolean visible)` - shows or hides the `Component`

Detailed information about using these methods is available in the LWT API documentation.

Using Components

Once you have your **`ComponentScreen`** defined, you can now start adding LWT components to it. All of the components defined by LWT are used alike, so the methods shown here are applicable to all of the LWT components. This example creates a screen with three buttons on it:

```
/**
* Demo Screen for the Button Component
**/

class ButtonScreen extends ComponentScreen {
   public ButtonScreen() {
      Button b1 = new Button("Button");
      add(b1);

      Button b2 = new Button("Large Button");
      b2.setRightEdge(Component.SCREEN_RIGHT, 0);
      b2.setBottomEdge(Component.HEIGHT,
      b2.getPreferredHeight() * 2);
      add(b2);

      Button b3 = new Button("Disabled Button");
      b3.setEnabled(false);
      add(b3);
   }
}
```

This code also shows how to modify the defaults when creating a new component. Since buttons are automatically created as 'Enabled', button b3 has specific code to initialize it as disabled. The code for button b2 demonstrates a method for changing the default size of the button.

The ComponentListener Interface

The **ComponentListener** interface is implemented by any class that wishes to receive events from a **Component**. The events vary depending on the object that originates the event, but can include events such as button actuation, checkbox selection, etc.

ComponentListener Interface Definition

This interface is defined as

- public interface ComponentListener

ComponentListener Interface Methods

The **ComponentListener** interface defines a single method:

- `ProcessComponentEvent()` – It processes an event received by a **Component**. It is defined as public void processComponentEvent(java.lang.Object source, int eventType) where source is the object which originated the event, and eventType is the type of event that occurred. EventType is defined by the originating object's class, and is only defined to be unique within that class. A listener should call the appropriate methods in the originating object to determine information about the event that occurred.

The InteractableComponent Class

The **InteractableComponent** class is a subclass of **Component**. This class adds functionality to allow it to interact with the user, i.e. the user can 'press' and 'release' objects created from this class. It provides the basic pointer and key event handling required by checkboxes, buttons, image labels, etc. It also provides methods for setting and getting the text and font associated with the component.

An **InteractableComponent** may be actuated by tapping and releasing within its bounds. It may also be actuated by pressing and releasing the devices 'Enter' key when it has key focus. **Button**, **Checkbox**, and **ImageLabel** are all subclasses of **InteractableComponent**.

InteractableComponent Definition and Constructor

The **InteractableComponent** class is defined by:

```
public abstract class InteractableComponent extends
Component
```

Its only constructor is:

- `InteractableComponent(java.lang.String label)` – This constructs a new **InteractableComponent** with the specified label. It also sets acceptsKeyFocus to true so that this **Component** can accept focus and key events.

InteractableComponent Methods

Since this class extends the **`Component`** class, it inherits many fields and methods from that class. In addition, it defines the following methods:

- `abstract void componentActuated()` - called when the Component is actuated (tapped and released). Subclasses must define this method to perform the appropriate actions when they are actuated.
- `protected void dispatchComponentEvent(int event)` - dispatches the specified event to this InteractableComponent's listener, if any
- `javax.microedition.lcdui.Font getFont()` - gets the Font associated with this label
- `java.lang.String getLabel()` - gets the label for this Component
- `boolean isPressed()` - checks if this InteractableComponent is currently pressed
- `protected boolean keyPressed(int keyCode)` - called when a key is pressed
- `protected boolean keyReleased(int keyCode)` - called when a key is released
- `abstract void paint(javax.microedition.lcdui.Graphics g)` - renders the Component
- `protected void pointerDragged(int x, int y)` - called when the pointer is dragged
- `protected void pointerPressed(int x, int y)` - called when the pointer is pressed
- `protected void pointerReleased(int x, int y)` - called when the pointer is released
- `void setComponentListener(ComponentListener l)` - sets this InteractableComponent's listener
- `void setFont(javax.microedition.lcdui.Font font)` - sets the Font object for rendering the label, if any
- `void setLabel(java.lang.String label)` - sets the label for this InteractableComponent
- `void setPressed(boolean b)` - sets the pressed/released state of this Component

The Button Class

A **Button** is a subclass of **InteractableComponent**. As such, it can interact with the user. Optionally, it can display a label to convey its function.

Button Class Definition and Constructors

The **Button** class is a subclass of **InteractableComponent** and is defined by:

```
public class Button extends InteractableComponent
```

It has two constructors:

- Button() – Construct a Button with no label
- Button(java.lang.String label) – Construct a Button with the given text string as a label

Button Class Fields

The **Button** class has a single constant, which is used to send an event to a **Button's** listener, if any, when the **Button** is pressed.

- public static int BUTTON_ACTION_EVENT

Button Class Methods

The **Button** class inherits methods from **InteractableComponent** and **Component**. In addition, it defines these methods:

- `void componentActuated()` – called when this Button is actuated. This implementation dispatches a BUTTON_ACTION_EVENT to the Button's listener, if any.
- `int getPreferredHeight()` – gets the preferred height for this Button
- `int getPreferredWidth()` – gets the preferred width for this Button
- `void paint(javax.microedition.lcdui.Graphics g)` – renders the Button

An example of the **Button** class is described in Using Components.

The ImageLabel Class

ImageLabel is a **Component** that can display an image and/or a text label. The image and/or text label, referred to as the **ImageLabel's** contents, can be collectively placed North, South, East, West, or centered within the bounds of the **ImageLabel**. See ImageLabel for more information.

The relative layout of the contents can be controlled by specifying the location of the text label relative to the image. The text label can be placed either above the image, below the image, to the left of the image, to the right of the image, or in the center of the image.

An **ImageLabel** may be either "interactable" or non-"interactable". An "interactable" **ImageLabel** may be actuated by the user and its state can be normal, disabled or pressed. Separate images may be provided for each of these states; the normal image is used by default if a specific image is not provided for a given state.

A non-interactable **ImageLabel** cannot be actuated by the user; its state can be either normal or disabled.

The separation between the text and the image when the LABEL_RIGHT or LABEL_LEFT position is selected is 3 pixels. The separation between the text and the image when the LABEL_ABOVE or LABEL_BELOW position is selected is 2 pixels. In the absence of either text or image, the separation will be zero.

ImageLabel Class Definition and Constructors

The **ImageLabel** class is a subclass of **InteractableComponent** and is defined by:

```
public class ImageLabel extends InteractableComponent
```

This class has two constructors:

- `ImageLabel(javax.microedition.lcdui.Image normal, javax.microedition.lcdui.Image disabled, javax.microedition.lcdui.Image pressed, java.lang.String label)` – constructs a new interactable ImageLabel with the specified images for the three states (normal, disabled and pressed) and the specified label.
- `ImageLabel(javax.microedition.lcdui.Image normal, javax.microedition.lcdui.Image disabled, java.lang.String label)` – constructs a new non-interactable ImageLabel with the specified images for the two states (normal and disabled).

ImageLabel Class Fields

The fields defined for the **`ImageLabel`** class are mainly used to indicate the relative positions of the image and text items within the object. One event is defined for use by any listeners.

- `static int ALIGN_CENTER` - the image and label should be horizontally and vertically centered within the ImageLabel
- `static int ALIGN_EAST` – The image and label should be vertically centered and aligned with the right edge of the ImageLabel
- `static int ALIGN_NORTH` – The image and label should be horizontally centered and aligned with the top edge of the ImageLabel
- `static int ALIGN_SOUTH` – The image and label should be horizontally centered and aligned with the bottom edge of the ImageLabel
- `static int ALIGN_WEST` – The image and label should be vertically centered and aligned with the left edge of the ImageLabel
- `static int HORIZONTAL_GAP` - the horizontal gap between image and the text
- `static int IMAGE_LABEL_ACTION_EVENT` – event indicating that the ImageLabel was actuated by the user (for interactable ImageLabels only)
- `static int LABEL_ABOVE` – the label, if any, should be placed above the image and horizontally centered relative to the image
- `static int LABEL_BELOW` – the label, if any, should be placed below the image and horizontally centered relative to the image
- `static int LABEL_CENTER` – the label, if any, should be centered on the image
- `static int LABEL_LEFT` – the label, if any, should be placed to the left of the image and vertically centered relative to the image
- `static int LABEL_RIGHT` – the label, if any, should be placed to the right of the image and vertically centered relative to the image
- `static int TRANSPARENT` - transparent background color
- `static int VERTICAL_GAP` - the vertical gap between image and text

ImageLabel Class Methods

The **`ImageLabel`** class inherits many methods from **`InteractableComponent`** and **`Component`**. In addition, it defines the following methods:

- `void componentActuated()` - called when this `ImageLabel` is actuated by the user.

- `int getBackgroundColor()` - gets the current background color.
- `int getForegroundColor()` - gets the current foreground color.
- `int getPreferredHeight()` - gets the preferred height of the `ImageLabel.`
- `int getPreferredWidth()` - gets the preferred width of the `ImageLabel.`
- `void paint(javax.microedition.lcdui.Graphics g)` - paints this `ImageLabel.`
- `void setAlignment(int alignment)` - sets the desired alignment for this `ImageLabel.`
- `void setBackgroundColor(int color)` - sets the background color.
- `void setDisabledImage(javax.microedition.lcdui.Image i)` – sets the image for the disabled state.
- `void setForegroundColor(int color)` - sets the foreground color
- `void setLabelLocation(int location)` - sets the location of the label, if any, relative to the `ImageLabel's` image.
- `void setNormalImage(javax.microedition.lcdui.Image i)` - sets the image for the normal state.
- `void setPressedImage(javax.microedition.lcdui.Image i)` – sets the image for the pressed state.

Checkbox Class

The **`Checkbox`** is a component that represents a boolean value. A **`Checkbox`** can be used as-is to provide a single, independent choice for the user. A **`Checkbox`** can also be added to a **`CheckboxGroup`** to provide more extensive functionality.

Checkbox Class Definition and Constructors

The **`Checkbox`** class is a subclass of **`InteractableComponent`** and is defined by:

```
public class Checkbox extends InteractableComponent
```

It has two constructors:

- `Checkbox()` - creates a new Checkbox with an empty (null) label
- `Checkbox(java.lang.String label)` - creates a new Checkbox with the specified label

Checkbox Class Fields

These constants are used to pass events to a listener, if any, or to define the appearance of the **`Checkbox`**:

- `static int CHECKBOX_CHECKED_EVENT` - event indicating that this `Checkbox` was checked
- `static int CHECKBOX_UNCHECKED_EVENT` - event indicating that this `Checkbox` was unchecked
- `static int STYLE_CHECKBOX` - indicates that this `Checkbox` should look like a checkbox
- `static int STYLE_LIST_ITEM` - indicates that this `Checkbox` should look like a list item
- `static int STYLE_RADIO_BUTTON` - indicates that this `Checkbox` should look like a radiobutton

Checkbox Class Methods

`Checkbox` inherits several methods from **`InteractableComponent`** and **`Component`**.

In addition, it defines the following methods:

- `void componentActuated()` - called when this `Checkbox` is actuated by the user
- `void gainedFocus()` - called when this `Checkbox` gains key focus
- `int getPreferredHeight()` - gets the preferred height of this `Checkbox`
- `int getPreferredWidth()` - gets the preferred width of this `Checkbox`
- `boolean getValue()` - gets the current value of this `Checkbox`
- `void lostFocus()` - called when this `Checkbox` loses key focus
- `void paint(javax.microedition.lcdui.Graphics g)` - renders the `Checkbox`
- `void setValue(boolean value)` - sets the current value of this `Checkbox`.

Grouping Checkboxes

The **`Checkbox`** components are designed to be used together with other ones. Although individual **`Checkboxes`** have many uses, it's also nice to be able to combine

them to provide a multiple-choice grouping. The **CheckboxGroup** class is designed to make that possible. Here's how to use it:

```
/* Demo Screen for the RadioButton Component (Checkbox with
a
CheckboxGroup)
**/

class RadiobuttonScreen extends ComponentScreen {
   public RadiobuttonScreen() {
      CheckboxGroup g = new  CheckboxGroup
      (Checkbox.STYLE_RADIO_BUTTON);

      Checkbox c1 = new Checkbox("Radio Button A");
      add(c1);
      g.add(c1);

      c1 = new Checkbox("Radio Button B");
      add(c1);
      g.add(c1);
      c1 = new Checkbox("Radio Button C");
      add(c1);
      g.add(c1);

      Checkbox c2 = new Checkbox("Large Radio Button");
      c2.setRightEdge(Component.SCREEN_RIGHT, 0);
      c2.setBottomEdge(Component.HEIGHT,
      c2.getPreferredHeight() * 2);
      add(c2);
      g.add(c2);

      Checkbox c3 = new Checkbox("Small Radio Button");
      c3.setRightEdge(Component.SCREEN_RIGHT, 0);
      c3.setBottomEdge(Component.HEIGHT,
      c3.getPreferredHeight() - 8);
      add(c3);
      g.add(c3);

      Checkbox c4 = new Checkbox("Disabled Radio Button");
      c4.setEnabled(false);
      add(c4);
      g.add(c4);
   }
}
```

Note in this example that each **Component** is added not only to the **ComponentScreen**, but also to the **CheckboxGroup**. The type of **CheckboxGroup** is specified when the **CheckboxGroup** is instantiated (in this

case it's a radio button group), and the **Checkboxes** themselves are created and customized just as the **Buttons** in the earlier Button example.

The CheckboxGroup Class

The **CheckboxGroup** manages a group of **Checkboxes**. Unlike the AWT class of the same name, the use of a **CheckboxGroup** does not imply an exclusive list. A **CheckboxGroup** can be constructed for both exclusive and multiple selection modes. In LWT, the **CheckboxGroup** serves as a single reference point for several Checkboxes, eliminating the need to deal with each **Checkbox** individually. The **CheckboxGroup** supports a **ComponentListener**, so the interested object can listen to just the **CheckboxGroup**, rather than having to add itself as a listener to each of the Checkboxes individually.

CheckboxGroup Class Definition and Constructor

The **CheckboxGroup** class is a subclass of **Object**, and is defined by:

```
public class CheckboxGroup extends java.lang.Object
```

It has the following constructor:

- `public CheckboxGroup(int style)` throws `IllegalArgumentException` – creates a new CheckboxGroup with the given style. The argument `style` can be one of the following options: Checkbox.STYLE_CHECKBOX, Checkbox.STYLE_RADIO_BUTTON, or Checkbox.STYLE_LIST_ITEM.

CheckboxGroup Class Fields

The **CheckboxGroup** class defines the following constant:

- `public static final int CHECKBOXGROUP_SELECTION_CHANGED` – Event indicating that the value of one or more Checkboxes in this CheckboxGroup has changed. This constant has a value of 0x01.

CheckboxGroup Class methods

The **CheckboxGroup** class defines the following methods:

- `public int getSelectedIndex()` – Gets the index of the selected element. It returns the index of the selected element (or -1 if style is Checkbox.STYLE_CHECKBOX or the group has no Checkboxes);
- `public void setComponentListener(ComponentListener l)` – Sets this Component's listener. A Component can have only one listener at a time. The parameter `l` is a ComponentListener, or null if no listener is desired;
- `public int add(Checkbox b)` – Adds a Checkbox to this group and returns the index assigned to it. The parameter `b` is a non-null Checkbox to add;
- `public void insert(Checkbox b, int index)` – Inserts a Checkbox into this group at the specified index. If the index is less than 0, the Checkbox is inserted at the beginning of the list (index = 0). If the index is greater than the number of Checkboxes in this group, the Checkbox is appended at the end of the list. The parameter `b` is a non-null Checkbox to add, and the parameter `index` is the index where the Checkbox is to be inserted;
- `public void remove(Checkbox b)` – Removes the specified Checkbox from this group. This method does nothing if the specified Checkbox is null or not currently added to this CheckboxGroup. For single select groups, the selected Checkbox defaults to index 0 if the Checkbox to be removed is currently selected. The parameter `b` is the Checkbox to be removed from this list;
- `public Checkbox getCheckbox(int index)` throws `IndexOutOfBoundsException` – Gets the Checkbox with the specified index. The parameter `index` is the index of the Checkbox.
- `public int getCheckboxCount()` – Gets the number of Checkboxes that belong to this CheckboxGroup;
- `public void remove(int index) throws IndexOutOfBoundsException` – Removes the Checkbox with the specified index from this group. For single select groups, the selected Checkbox defaults to index 0 if the Checkbox to be removed is currently selected;
- `public boolean isSelected(int index)` throws `IndexOutOfBoundsException` – Gets the value of the Checkbox with the specified index. The parameter `index` is the index of the Checkbox. It returns `true` if the Checkbox is selected, otherwise `false;`
- `public void setSelectedFlags(boolean[] selectedArray)` throws `llegalArgumentException`, `NullPointerException` – Sets the values of the group's Checkboxes to the values of the provided array. The number of elements in the array must be greater than or equal to the number of Checkboxes.

For multiple-select CheckboxGroups (Checkbox.STYLE_CHECKBOX), this method sets the value of every Checkbox; an arbitrary number of elements may be selected. For single-select CheckboxGroups (Checkbox.STYLE_RADIO_BUTTON or Checkbox.STYLE_LIST_ITEM), exactly one array element must have the value `true`. If no element is `true`, the first Checkbox will be set to `true`. If two or more elements are `true`, only the first `true` element will be recognized; the other elements will be ignored. This method has no effect if the CheckboxGroup contains no Checkboxes.

- `public void setSelectedIndex(int index, boolean value)` throws `IndexOutOfBoundsException` – Sets the selection for this CheckboxGroup. For multiple-select CheckboxGroups (Checkbox.STYLE_CHECKBOX), this method simply sets the value of the specified Checkbox to the specified value. For single-select CheckboxGroups (Checkbox.STYLE_RADIO_BUTTON or Checkbox.STYLE_LIST_ITEM), this method sets the specified Checkbox provided the specified value is true; otherwise, the call is ignored. The parameter `index` is the index of the checkbox to set or select, and `value` is `false` for a new value of the checkbox for multi-select lists, or `true` for single-select lists.

The TextComponent Class

The **`TextComponent`** class is the base class for **`TextArea`** and **`TextField`**. It provides common functionality such as text manipulation, font control, length limiting, constraints, justification and echo character support. The native implementation of **`TextComponents`** may include support for selection, cut/copy/paste, handwriting recognition, keypad prediction, etc.; however, these features are not exposed in the API since they are not guaranteed to be supported by all devices.

TextComponent Class Definition and Constructor

The **`TextComponent`** class is a subclass of **`Component`**, and is defined by:

```
public abstract class TextComponent extends Component
```

There is no constructor for this class. Use a **`TextArea`** or **`TextField`** class to define a text-handling object.

TextComponent Class Fields

The **`TextComponent`** class defines the following constants:

- `static int JUSTIFY_CENTER` - constant for center justification
- `static int JUSTIFY_LEFT` - constant for left justification
- `static int JUSTIFY_RIGHT` - constant for right justification

- `static int NO_LIMIT` - constant for no length limit

TextComponent Methods

The **TextComponent** inherits several methods from **Component**. In addition, it defines these methods:

- `void appendChar(char c)` - appends the specified character at the end of the current text
- `void appendText(java.lang.String text)` - appends the specified text at the end of the current text
- `int getConstraint()` - gets the text entry constraint for this `TextComponent`
- `char getEchoChar()` - obtains the echo character used by this `TextComponent`, or 0 if the actual characters are displayed.
- `javax.microedition.lcdui.Font getFont()` - gets the font currently used by this `TextComponent`
- `int getLengthLimit()` - gets the length limit
- `int getPreferredHeight()` - gets the preferred height of this `TextComponent`
- `int getPreferredWidth()` - gets the preferred width of this `TextComponent`
- `java.lang.String getText()` - obtains the text contained in this `TextComponent`
- `void insertChar(char c, int index)` - inserts the specified character at the specified index in the current text
- `void insertText(java.lang.String newText, int index)` – inserts the specified text at the specified index in the current text
- `boolean isEditable()` - checks whether or not the contents of this `TextComponent` may be edited by the user
- `boolean keyPressed(int keyCode)` - called when a key is pressed
- `boolean keyRepeated(int keyCode)` - called when a key is repeated (held down)
- `void paint(javax.microedition.lcdui.Graphics g)` - renders the `Component`
- `void pointerPressed(int x, int y)` - called when the pointer is pressed within this `Component`
- `void setConstraint(int constraint)` - sets the text entry constraint for the contents of this `TextComponent`

- `void setEchoChar(char c)` - sets the echo character to be displayed by this `TextComponent`. Use 0 to display the actual characters typed.
- `void setEditable(boolean editable)` - sets whether or not the `TextComponent` can be edited by the user
- `void setFont(javax.microedition.lcdui.Font newFont)` - sets the current font of the `TextComponent`
- `void setJustification(int justification)` - sets the justification of this `TextComponent`
- `void setLengthLimit(int maxChars)` - sets the length limit
- `void setText(java.lang.String newText)` - sets the contents of the `TextComponent` to the specified `String`
- `protected void textChanged(int start, int end, boolean user)` - called whenever the contents of this `TextComponent` are changed, either by the user or programmatically

The TextField Class

The **`TextField`** class defines a single line **`TextComponent`** that scrolls horizontally as needed.

TextField Class Definition and Constructor

This class is defined by

```
public class TextField extends TextComponent
```

The only constructor is:

- `TextField(java.lang.String text, int columns)`

This constructor constructs a new **`TextField`** with the given text and number of columns. The number of columns is used only to establish the preferred width for layout purposes; it does not restrict the length of text that can be entered into the **`TextField`**.

TextField Class Methods

The **`TextField`** class inherits all of its methods from the **`TextComponent`** and **`Component`** classes. There are no additional methods defined in the **`TextField`** class.

The TextArea Class

The **TextArea** class is a multi-line **TextComponent** that scrolls vertically as needed. Scrolling is automatically provided by the platform. A scrollbar (or other similar mechanism) is provided by the native UI as needed so that the user can adjust the scroll offset.

TextArea Class Definition and Constructor

This class is defined by

```
public class TextArea extends TextComponent
```

The only constructor is:

- `TextArea(java.lang.String text, int rows, int columns)`

It constructs a new **TextArea** with the given text, number of rows, and number of columns. The number of rows and columns is used only to establish the preferred height and width for layout purposes; it does not restrict the length of text that can be entered into the **TextArea**.

TextArea Class Methods

The **TextArea** class inherits all of its methods from the **TextComponent** and **Component** classes. There are no additional methods defined in the **TextArea** class.

The Slider Class

The **Slider** component represents a variable (and possibly adjustable) numeric value.

The **Slider** can either be a read-only device to display a value, or it can be an interactable device that allows the user to view and adjust a value.

Only horizontal **Slider** is supported since the vertical direction keys are reserved for changing key focus.

A **Slider**'s value can range from 0 to its maximum value, inclusive. The maximum value must be at least 0. There is no upper limit on the maximum value; however, the resolution of a **Slider** will be reduced if its maximum value exceeds its width.

Slider Class Definition and Constructor

The **`Slider`** class is a subclass of **`Component`**, and is defined by:

```
public class Slider extends Component
```

The constructor for a **`Slider`** is:

- `Slider(boolean interactive, int maxValue, int value)`

Slider Class Fields

The **`Slider`** class defines two fields which are passed as events to any listeners that may be active on the class. These fields are:

- `static int SLIDER_DRAGGED` - event indicating that the **`Slider'`**`s` value has been changed and that it is still being interacted with by the user. Since this event may happen repeatedly and quickly, the code that deals with it should execute quickly.
- `static int SLIDER_SET` - event indicating that the **`Slider'`**`s` value has been changed and that the user is no longer interacting with it.

Slider Class Methods

The **`Slider`** class inherits many methods from **`Component`**. In addition, it defines the following methods:

- `int getMaxValue()` - gets the maximum value of the `Slider`
- `int getPreferredHeight()` - gets the preferred height of the `Slider`
- `int getPreferredWidth()` - gets the preferred width of the `Slider`
- `int getValue()` - gets the current value of the `Slider`
- `protected boolean keyPressed(int keyCode)` - called when a key is pressed
- `protected boolean keyReleased(int keyCode)` - called when a key is released
- `protected boolean keyRepeated(int keyCode)` - called when a key is repeated (held down)
- `void paint(javax.microedition.lcdui.Graphics g)` - renders the `Component`

- `protected void pointerDragged(int x, int y)` - called when the pointer is dragged
- `protected void pointerPressed(int x, int y)` - called when the pointer is pressed
- `protected void pointerReleased(int x, int y)` - called when the pointer is released
- `void setComponentListener(ComponentListener l)` - sets this `Slider's` listener
- `void setMaxValue(int maxValue)` - sets the maximum value of the `Slider`
- `void setValue(int value)` - sets the current value of the `Slider`

7
Record Management System (RMS)

Overview

The most common mechanism for persistently storing data on a MIDP device is through RMS. RMS provides the capability to store variable length records on the device. Those records are accessible to any MIDlet in the MIDlet Suite, but not to MIDlets outside of the MIDlet Suite. The RMS implementation of the Motorola A830 handset is MIDP compliant, so there are no significant additions or changes to the MIDP specification.

Class Description

The API for the RecordStore is located in the package `javax.microedition.rms.`

Code Examples

The following simple code example will open the **`RecordStore`**. If any exception occurs it will be caught.

```
try {
   System.out.println("Opening RecordStore " + rsName + "
   ...");
   //try to open a record Store
   recordStore = RecordStore.openRecordStore(rsName, true);
   //keep a note for the last modified time for record
   store
   Date d = new Date(recordStore.getLastModified());
   System.out.println(recordStore.getName()+"modified last
```

```
            time: " +
             d.toString());
            }
            catch (RecordStoreException rse) {
            //process the IOException
        }
```

Tips

It is much faster to read and write in big chunks than it is to do so in small chunks.

Whenever you close a **RecordStore,** the close command will not return until all the pending writes have been written. A successful call to close a **RecordStore** guarantees that the data got written. It is then safe to power off the phone; a side effect to this is that the close command may take a while to return. Therefore, if a **RecordStore** is opened and closed for every write performance will be greatly affected.

Caveats

The maximum number of **RecordStore**s that the Motorola A830 handset supports depends on the number of files installed. Once the phone has 500 RecordStores (that includes resource files, wall papers, ring tones, and other files), then it will not be able to make more.

Therefore, if a MIDlet is to have many images, such as sprites used in animations, it may be advantageous to have them all in one image file and use clipping to display only what you need.

RecordStore can be of any size as long as there is file space available. A zero byte **RecordStore** is also allowed.

8
J2ME Networking

Overview

The J2ME platform on the Motorola A830 handset provides a variety of networking functionalities beyond those specified in MIDP. The additional networking protocols are added through the Generic Connection Interface in order to simplify the interface to the application as well as to reduce the need for additional classes. Most of the additional network connections are invoked using a runtime parameter similar to HTTP, reducing the learning curve for developers as well as the reducing potential application porting efforts. The following is a list of networking features for the Motorola A830 handset:

- HTTP
- HTTPS
- TCP Sockets
- SSL Secure Sockets
- UDP Sockets
- Serial Port Access

The standard networking protocol specified in MIDP 1.0 is HTTP. Although HTTP is useful and flexible for most data exchanges, many of the applications fall outside the standard request/response models of most browsers. Applications such as games and stock tickers require networking protocols with different characteristics. In order to accommodate these types of applications with reasonable efficiency, additional protocol stacks including UDP, TCP Sockets, SSL, and HTTPS have been added. These added networking functionalities not only provide the application developer with more communication options, it alleviates the need to perform inefficient workarounds for a strict HTTP environment. Other applications may also choose to take advantage of the bottom connector on the devices. The bottom connector is a serial port enabling communication with a variety of other devices. The Motorola A830 handset also provides serial port access through the Generic Connection Framework in order to provide applications a means to communicate to external devices such as GPS, OBD, PCs, etc.

Class Descriptions

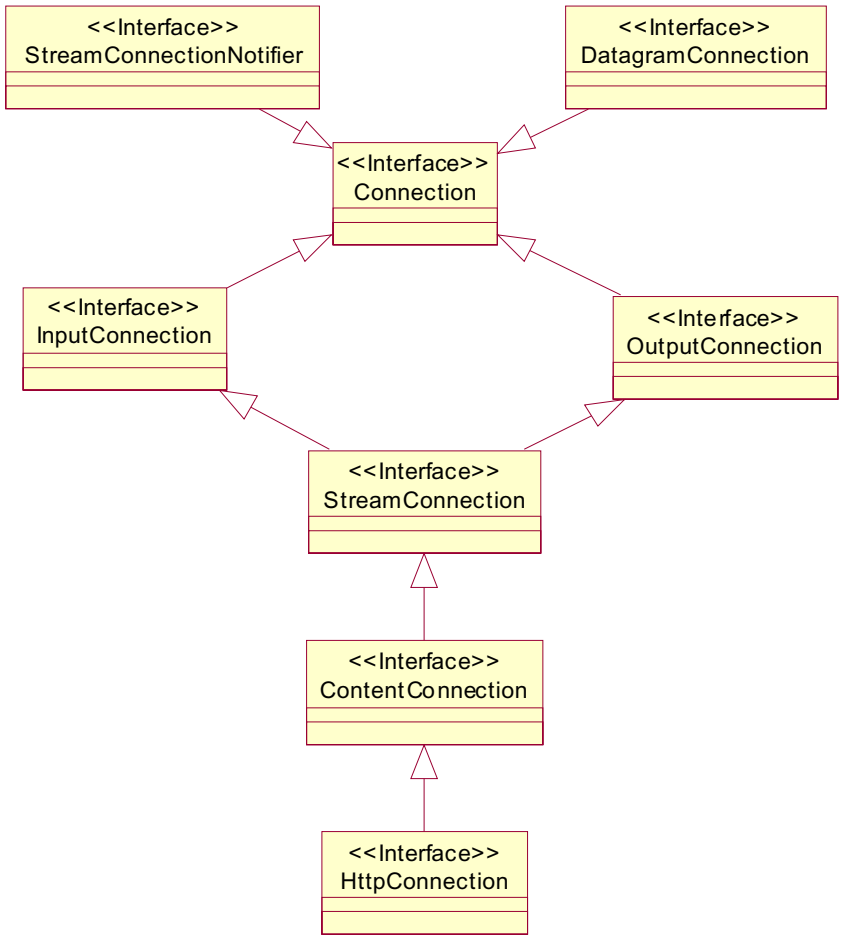

Figure 17. The Connection Framework

Since all the additional communication protocols have been added to the Generic Connection Framework, the access methods and parameters are very similar. The main calls are to the **`Connector`** class, which provides three static methods that accept different compile time parameters. The commonality between the three static methods is the first parameter in their signatures. This particular runtime parameter accepts **`Strings`** formatted in the standard Uniform Resource Locator format. The following is the list of method signatures:

- `Connector.open(String URL)` – default READ_WRITE, no timeout.
- `Connector.open(String URL, int mode)` - defaults to no timeout.
- `Connector.open(String URL, int mode, Boolean timeout)`

- `String URL` – parameter string describing the target conforms to the URL format as described in RFC 2396 for all networking protocols except for Serial Port.

- `int mode` – READ/WRITE/READ_WRITE

- `boolean timeout` – An optional third parameter, protocol may throw an IOException when it detects a timeout condition.

The timeout period for the TCP implementation on the Motorola A830 handset is 20 seconds on read operation and about 45 seconds on write operation if the timeout flag is set to true. If the timeout flag is set to false, the timeout time is indefinite. The lingering time for closing sockets is 0 second (if the socket closed by the server the lingering time will be less than 100 ms). If a new socket is requested within this time frame and the maximum number of sockets opened has been reached (4 sockets), then an IOException is thrown.

Applications requesting a network resource for any protocol must use one of the three methods above. The URL is the distinguishing argument that determines the difference between HTTP, Serial, etc. The following chart details the prefixes that should be used for the supported protocols.

Table 2 - Supported Protocols on the Motorola A830 handset

Protocol	URL Format
HTTP	`http://`
HTTPS	`https://`
TCP Sockets	`socket://`
SSL Secure Sockets	`securesocket://`
UDP Sockets	`datagram://`
Serial Port	`comm:<Port_Name>;`

- <Port_Name> - should be derived from the return string of System.get Property("serialport.name").

HTTP

The HTTP implementation follows the MIDP 1.0 standard. The `Connector.open()` methods return a **`HttpConnection`** object that is then used to open streams for reading and writing. The following is a code example:

```
HttpConnection hc = (HttpConnection)Connector.open(
“http://www.motorola.com”);
```

In this particular example, the standard port 80 is used, but this parameter can be specified as shown in the following example:

```
HttpConnection hc = (HttpConnection)Connector.open(
“http://www.motorola.com:8080”);
```

The other static Connector methods work in the same manner, but they provide the application additional control in dealing with the properties of the connection. By default, HTTP 1.1 persistency is used to increase efficiency while requesting multiple pieces of data from the same server. In order to disable persistency, set the “Connection” property of the HTTP header to “close”.

HTTPS

The HTTPS implementation follows the MIDP 1.0 standard, save for the security aspects. The `Connector.open()` methods return a **`HttpConnection`** object that is then used to open streams for reading and writing. The following is a code example:

```
HttpConnection hc = (HttpConnection)Connector.open(
“https://www.motorola.com”);
```

In this particular example, the standard port 443 is used, but this parameter can be specified as shown in the following example:

```
HttpConnection hc = (HttpConnection)Connector.open(
“https://www.motorola.com:8888”);
```

The other static **`Connector`** methods work in the same manner, but they provide the application additional control in dealing with the properties of the connection.

Due to memory constrain, Motorola A830 handset can support only one secure session (i.e. if other application like Browser already opened a secure socket, a MIDlet application will get an IOException when it tries to open only one connection which is HTTPS connection).

Note – Only Verisign Certificates are supported in the Motorola A830 handset. The following is a list of supported features:

- SSL 3.0 (Secure Socket Layer)
- TLS 1.0 (Transport Layer Security)
- Server Authentication

TCP Sockets

The low-level socket used to implement the higher-level HTTP protocol is exposed to applications via the Generic Connection Framework. The usage is similar to the examples above, however, a **`StreamConnection`** is returned by the `Connection.open()` method, as shown in the following example:

```
StreamConnection sc =
      (StreamConnection)Connector.open(
      “socket://www.motorola.com:8000”);
```

Although similar to HTTP, notice the required port number at the end of the remote address. In the previous protocols, those ports are well known and registered so they are not required, but in the case of low level sockets, this value is not defined. The port number is a required parameter for this protocol stack.

SSL Secure Sockets

The low-level socket used to implement the higher-level HTTPS protocol is also exposed to applications via the Generic Connection Framework. The usage is similar to the examples above.

```
StreamConnection sc = (StreamConnection)Connector.open(
“securesocket://www.motorola.com:8000”);
```

As with non-secure sockets, the port number is a required parameter for this protocol stack.

UDP Sockets

If networking efficiency is of greater importance than reliability, datagrams (UDP) sockets are also available to the application in much the same manner as other networking protocols. The **`Connector`** object in this case returns a **`DatagramConnection`** object, as is shown in the following example:

```
DatagramConnection dc =
(DatagramConnection)Connector.open(
“datagram://170.169.168.167:8000”);
```

Much like low-level sockets, accessing UDP requires both a target address and a port number. The Motorola A830 handset supports a maximum outgoing and incoming payload of 1472 bytes and 2944 bytes, respectively.

Serial Port Access

Applications utilizing the bottom connector (serial port) to communicate with a variety of devices are given exclusive access to the port until either the application voluntarily releases the port or is terminated. Much like any other networking connection, opening a serial port is not guaranteed and an exception can be thrown. If another application native or Java is using the port, or a cable is not attached to the device, an IOException may be thrown. In the normal usage scenario, the **`Connector`** object in this instance returns a **`StreamConnection`**, as is shown in the following example:

```
String port_name= System.getProperty("serialport.name");
String max_baudrate=
```

```
System.getProperty("serialport.maxbaudrate");
if(baudrate > max_baudrate) baudrate= max_baudrate;
StreamConnection sc = (StreamConnection)Connector.open(
“comm:” + ”port_name” + ”;baudrate=” + baudrate +
”;parity=n;databits=8;stopbits=1;flowcontrol=n/n”);
```

Although serial port access is integrated into the Generic Connection Framework, the URL parameters passed in deviates from the other networking protocols. The optional parameters, such as baud rate, parity, etc are appended to the base parameter of “comm:0”. Optional parameters are listed below along with the default values when not explicitly specified:

Table 3 - Connection Optional Parameters

Parameter	Syntax	Options	Default
baudrate	`baudrate = x`	`[300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200]`	`192000`
databits	`databits = x`	`[8,7]`	`8`
stopbits	`stopbits = x`	`1, 1,5 and 2`	`1`
parity with mapping	`parity = x`	`[n,o,e,s,m] n=none, o=odd, e=even, s=space, m=mark`	`n`
Flow control	`flowcontrol = outflow/inflow`	`[n, s, h] / [n, s, h] n=none, s=software,h=hardware`	`N/n`
`autocts`	`autocts= x`	`on or off`	`off`
`autorts`	`autorts= x`	`on or off`	`off`
`blocking`	`blocking = x`	`on or off`	`off`

Note - The following combinations of properties are not supported.

- 7 databits with none parity
- 8 databits with mark parity
- 8 databits with space parity
- 8 databits with odd parity
- 8 databits with even parity

IOException will be thrown while trying to use any of the unsupported combinations in Connector.open().

All properties must be semicolon separated. If not all properties are passed; the remaining properties will be taken as default values. The order of properties in the argument does not matter.

```
name = "comm:0;baudrate=38400;"
```

Here, the flow control, parity, data bits and stop bits will use the default values.

For mode and timeout refer to the CLDC API specification for the Connector class.

Communicating on a Port

The open method of the **`Connector`** class returns a **`StreamConnection`** object for the serial port. **`StreamConnection`** has methods for obtaining input and output streams from a port. The base interface, **`Connection`**, has a method to close the port. (Refer to the class hierarchy from **`StreamConnection`** from J2ME CLDC API specification).

There are five basic steps to communicating with a port:

- Open the port using the `open()` method of **`Connector`**. If the port is available, this returns a **`StreamConnection`** object for Comm port. Otherwise, an IOException is thrown.
- Get the output stream using the openOutputStream() method of **`OutputConnection`**.
- Get the input stream using the openInputStream() method of **`InputConnection`**.
- Read and write data onto those streams.
- Close the port using the `close()` method of both the **`Connection`** and open Streams.

Once the connection has been established, simply use the normal methods of any input or output stream to read and write data. The openInputStream and openOutputStream methods of **`StreamConnection`** are similar to the methods of the socket **`StreamConnection`**.

Example using StreamConnection

`Connector.open` is used to open the serial port and a **`StreamConnection`** is returned. From the **`StreamConnection`** the `InputStream` and `OutputStream` are opened. It is used to read and write every character until the connection is closed(-1). If an exception is thrown the connection and stream are closed.

```
StreamConnection sc = null;
InputStream is = null;
OutputStream os = null;

/*
 * Create the parameter String with options specified
 */
String parameter =
"comm:0;baudrate=19200;parity=n;databits=8;stopbits=1;
flowcontrol=n/n”;

try{
   sc = (StreamConnection)Connector.open(parameter,
```

```
            Connector.READ_WRITE, false);
            os = sc.openOutputStream();
         is = sc.openInputStream();
         int ch;
         while ((ch = is.read() ) != -1) {
               os.write(ch);
         }
      } finally {
         if (sc != null)
            sc.close();
         if(is != null)
            is.close();
         if(os != null)
               os.close();
      }
```

Implementation Notes

As stated in the previous sections, the Motorola A830 handset supports some networking options. The networking options however are limited by both memory and bandwidth, which place hard restrictions on the applications. These limitations manifest themselves mainly in the number of simultaneous connections that can be opened.

Maximum number of sockets is 4 of any combinations of HTTP, HTTPS, socket, securesocket and UDP. Due to memory constrain A830 can support only one secure session (i.e. if other application like Browser already opened a secure socket, a KJava midlet will get an IOException when it tries to open only one connection which is HTTPS connection). If the maximum number of sockets is concurrently opened by the application and a fiveth socket is requested, an exception is thrown to the calling application.

Only one serial port is available. Any attempts to open 2 concurrent serial port connections results in a thrown exception.

Tips

A factor to take into consideration while developing networked applications is the blocking nature of many javax.microedition.io and java.io object methods. It is advisable to spawn another thread specifically dedicated to retrieving data in order to keep the user interface interactive. If a single thread is used to retrieve data on a blocking call, the user interface becomes inactive with the end-user perceiving the application as “dead”.

Reading from an **`InputStream`** using an array is faster then reading byte by byte, when the length of the data is known. For example, if the content length is provided in the header of the **`HttpConnection`**, then an array of the specified size can be used to read the data.

The **`InputStream`** and **`OutputStream`** as well as the **`Connection`** object need to be completely closed.

An application in the paused state can still continue to actively use the networking facilities of the Motorola A830 handset.

The platform does not support simultaneous voice and data transmissions.

9 Gaming

Functional Description

The Gaming API provides gaming related functionality to J2ME MIDlet writers. This functionality includes the ability and support for transparent images, the ability to play simple sounds and sound effects during a game, the ability to detect simultaneous key presses, support for sprites, and support for dynamically changing the palette color associated with an image.

Class Hierarchy

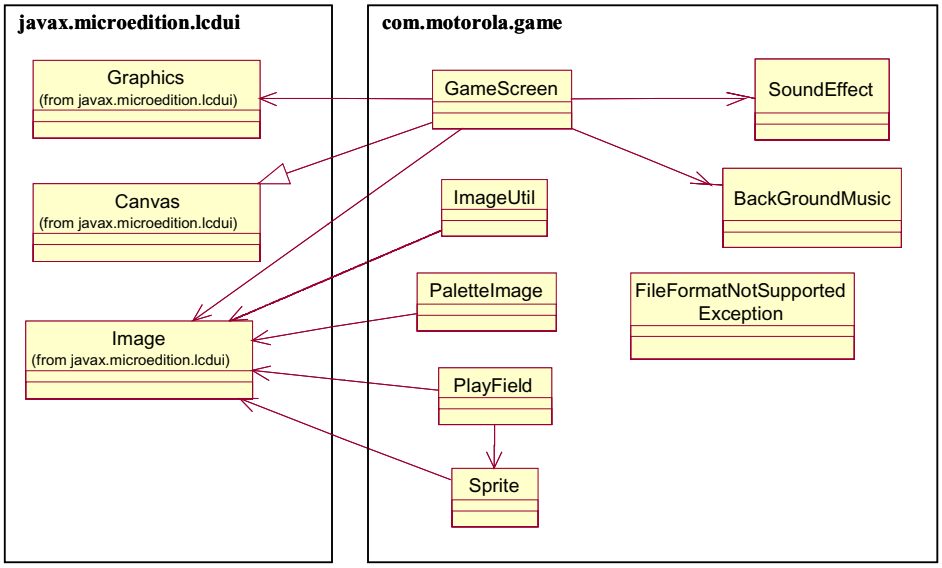

Figure 18. Gaming API class hierarchy

BackgroundMusic Class

The **`BackgroundMusic`** class encapsulates the data for a game's background music. A game may create several **`BackgroundMusic`** objects, but only one may be playing at any one time. The sound data may be stored on the device as a named resource in the application JAR file, or it can be stored on a server and retrieved via the network. **`BackgroundMusic`** is played by a **`GameScreen`**.

BackgroundMusic Methods

The **`BackgoroundMusic`** class defines the following methods:

- `public static BackgroundMusic createBackgroundMusic(java.lang.String name)` throws `FileFormatNotSupportedException` – Creates a BackgroundMusic for the sound data stored in the specified named resource or URL.

Using BackgroundMusic

Example:

```
BackgroundMusic bgm1 =
BackgroundMusic.createBackgroundMusic("/FunkyTunes.bgm");
BackgroundMusic bgm2 =
BackgroundMusic.createBackgroundMusic("http://www.motorola.c
om/sounds/JazzyTunes.bgm");
```

GameScreen Class

The **`GameScreen`** class provides the basis for a game user interface. In addition to the features inherited from MIDP's **`Canvas`** (commands, input events, etc.) it also provides game-specific capabilities such as an off-screen buffer with synchronized flushing and key status polling. A game may provide its own thread to run the game loop. A typical loop will check for input, implement the game logic, and then render the updated user interface.

GameScreen Fields

The **`GameScreen`** class defines the following fields:

- `public static final int DOWN_KEY` - The bit representing the DOWN key. This constant has a value of 0x02.
- `public static final int FIRE_KEY` - The bit representing the FIRE key. This constant has a value of 0x10.

- `public static final int GAME_A_KEY` - The bit representing the GAME_A key (may not be supported on all devices). This constant has a value of 0x20.
- `public static final int GAME_B_KEY` - The bit representing the GAME_B key (may not be supported on all devices). This constant has a value of 0x40.
- `public static final int GAME_C_KEY` - The bit representing the GAME_C key (may not be supported on all devices). This constant has a value of 0x80.
- `public static final int GAME_D_KEY` - The bit representing the GAME_D key (may not be supported on all devices). This constant has a value of 0x100.
- `public static final int LEFT_KEY` - The bit representing the LEFT key. This constant has a value of 0x04.
- `public static final int RIGHT_KEY` - The bit representing the RIGHT key. This constant has a value of 0x08.
- `public static final int UP_KEY` - The bit representing the UP key. This constant has a value of 0x01.
- `public static final int PRIORITY_MAX` - The maximum priority for playing sound effects. This constant has a value of 100.
- `public static final int PRIORITY_MIN` - The minimum priority for playing sound effects. This constant has a value of 0.
- `public static final int VOLUME_MAX` - The maximum volume for playing sound effects. This constant has a value of 100.
- `public static final int VOLUME_MIN` - The minimum volume for playing sound effects. This constant has a value of 0.

GameScreen Methods

The **`GameScreen`** class defines the following methods:

- `protected Graphics getGraphics()` - Obtains the Graphics object for rendering GameScreens. The Graphics object renders to an off-screen buffer whose size is equal to that of the GameScreen (use getWidth() and getHeight() to determine the size of the GameScreen). The buffer is initially filled with white pixels. Rendering operations do not appear on the display until flushGraphics() is called; flushing the buffer does not change its contents (that is, the pixels are not cleared as a result of the flushing operation). Only one image buffer is supported because without a vertical sync blanking period or its equivalent, there is little or no benefit from having multiple image buffers. Only one Graphics object exists for each GameScreen instance.
- `public int getKeyStates()` - Gets the states of the physical keys. Each bit in the returned integer represents a specific key on the device. A key's bit

will be set if the key is currently pressed or was pressed at least once since the last time this method was called. The bit will be 0 if the key is not currently pressed and was not pressed at all since the last time this method was called. This latching behavior ensures that a rapid key press and release will always be caught by the game loop, regardless of how slowly the loop runs. This method may be called twice to check if a key is currently pressed; that is, calling this method twice effectively disables the latching behavior. The lower bits are defined by UP_KEY, DOWN_KEY, LEFT_KEY, etc.; the remaining bits may be mapped to device-specific keys.

For example:

```
// Get the key state and store it
int keyState = gameScreenObject.getKeyStates();
if ((keyState & LEFT_KEY) != 0) {
   positionX--;
} else if ((keyState & RIGHT_KEY) != 0) {
   positionX++;
}
```

- `public void enableKeyEvents(boolean enabled)` - Enables or disables key event calls to this GameScreen. If disabled, the Canvas key event methods (keyPressed, keyRepeated, keyReleased) are not called when keys are pressed or released; however, the developer can still call getKeyStates to query the state of the keys. For games that poll key state and do not need event-driven key information, disabling key events can improve performance.
  Note that this setting is unique to each GameScreen instance; other GameScreens, when shown, are subject to their own setting for key events.

- `public void paint(javax.microedition.lcdui.Graphics g)` - Paints this GameScreen. By default, this method does nothing. It can be overridden according to application needs.

- `public void flushGraphics(int x, int y, int width, int height)` – Waits until the end of the current screen refresh cycle and then flushes the specified region of the off-screen buffer to the display driver. This method does not return until that region of the buffer has been completely flushed. The pixels of the off-screen buffer are not changed as a result of the flush operation. Upon returning from this method, the application may immediately begin to render the next frame using the same buffer.

- `public void flushGraphics()` - Waits until the end of the current screen refresh cycle and then flushes all of the off-screen buffer to the display driver. This method does not return until the entire buffer has been completely flushed. The pixels of the off-screen buffer are not changed as a result of the flush operation. Upon returning from this method, the app may immediately begin to render the next frame using the same buffer.

- `public static int getDisplayColor(int color)` throws `IllegalArgumentException` – Gets the color that will be displayed if the specified color is requested. This method enables the developer to check the manner in which RGB values are mapped to the set of distinct colors that the device can actually display. For example, with a monochrome device, this method will return

either 0xFFFFFF (white) or 0x000000 (black) depending on the brightness of the specified color.

- `public void playSoundEffect(SoundEffect se, int volume, int priority)` - Plays the specified SoundEffect. A GameScreen's sound effects are heard only while it is the visible screen. A device capability of playing SoundEffects can be found by using the method soundEffectsSupported(). The platform's ability to play several SoundEffects simultaneously can be found by using the method getMaxSoundsSupported(). The priority specified for each request determines which sound(s) are heard when the number of simultaneous sound requests exceeds the capabilities of the device.
- `public boolean soundEffectsSupported()` - Checks whether the underlying platform supports SoundEffects. It returns true if SoundEffects are supported.
- `public boolean backgroundMusicSupported()` - Checks whether the underlying platform supports BackgroundMusic. It returns true if BackgroundMusic is supported.
- `public int getMaxSoundsSupported()` - Queries the underlying platform's capability to play multiple SoundEffects simultaneously.
- `public void stopAllSoundEffects()` - Stops all the SoundEffects that are playing. Note that this method does not affect background music.
- `public void playBackgroundMusic (BackgroundMusic bgm, boolean loop)` - Plays the specified BackgroundMusic object from the beginning. This method first stops the current BackgroundMusic if any. Thus, this method may be used to start background music (by specifying a non-null BackgroundMusic object), restart the current background music (by specifying the same BackgroundMusic object), change the background music, or end the background music (by specifying null). The `loop` parameter is set to true if the BackgroundMusic is to repeat indefinitely. Otherwise, set to false.

Using GameScreen

The GameDemoScreen class uses the **`GameScreen`** class to provide a UI screen for a hypothetic game. GameDemoScreen is a subclass of **`GameScreen`** that implements runnable for running the main game loop thread.

```
class GameDemoScreen extends GameScreen implements Runnable{
    // ...
    public void run() {

        // Get the Graphics object for the
        // off-screen buffer
        Graphics g = getGraphics();
        while (true) {
            // Check user input and update
            // positions if necessary
```

```
            int keyState = getKeyStates();
            if ((keyState & LEFT_KEY) != 0) {
                sprite.move(-1, 0);
            }
            else if ((keyState & RIGHT_KEY) != 0) {
                sprite.move(1, 0);
            }
            // Draw the background
            g.drawImage(backgroundImage,0,0, Graphics.TOP
                  + Graphics.LEFT);
            // Draw the sprite on top of the background
            sprite.draw(g);
            // Flush the off-screen buffer
            flushGraphics();
        }
    }
    // ...
}
```

ImageUtil Class

`ImageUtil` provides static methods useful to the manipulation of **`Image`** objects. Specifically, it provides methods for setting and getting RGB values, and also provides the ability to create a scaled instance of an existing **`Image`**.

ImageUtil Fields

The **`ImageUtil`** class defines the following fields:

- `public static final int SCALE_AREA` - Area scaling method.
- `public static final int SCALE_REPLICATE` - Replicate scaling method.
- `public static final int SCALE_SMOOTH` - Smooth scaling method.

ImageUtil Methods

The **`ImageUtil`** class defines the following methods:

- `public static void getPixels(Image src, int x, int y, int width, int height, int[] rgbData)` throws `ArrayIndexOutOfBoundsException` – Gets RGB pixel data from the specified region of the source image. The data is stored in the provided *int* array in row-major order using the standard 24-bit color format (0xRRGGBB). Note that the

color information stored in the image may be subject to the capabilities of the device's display. The `rgbData` must be instantiated previously calling this method, according to pixel amount that the user is requiring to the method. The parameters are the following: `src` - the source Image to retrieve the pixel data from; `x` - the horizontal location of left edge of the region; `y` - the vertical location of the top edge of the region; `width` - the width of the region; `height` - the height of the region; `height` - the height of the region; and `rgbData` - the array in which the pixel data is to be stored.

- `public static void getPixels(Image src, int[] rgbData)` throws `ArrayIndexOutOfBoundsException` – Gets RGB pixel data from the entirety of the source image. The data is stored in the provided int array in row-major order using the standard 24-bit color format (0xRRGGBB). Note that the color information stored in the image may be subject to the capabilities of the device's display. The `rgbData` must be instantiated previously calling this method, according to pixel amount that the user is requiring to the method. The parameters are the following: `src` - the source Image to retrieve the pixel data from; and `rgbData` - the array in which the pixel data is to be stored.
- `public static void setPixels(javax.microedition.lcdui.Image dest, int x, int y, int width, int height, int[] rgbData)` throws `ArrayIndexOutOfBoundsException`, `IllegalArgumentException` – Sets RGB pixel data in specified region of the destination image. The data must be stored in the int array in row-major order using the standard 24-bit color format (0xRRGGBB). The method parameters are the following: `dest` - The mutable destination Image whose pixels will be set; `x` - The horizontal location of left edge of the region; `y` - The vertical location of the top edge of the region; `width` - The width of the region; `height` - The height of the region; and `rgbData` - The array of RGB pixel values.
- `public static void setPixels(javax.microedition.lcdui.Image dest, int[] rgbData)` throws `ArrayIndexOutOfBoundsException`, `IllegalArgumentException` – Sets RGB pixel data in the entirety of the destination image. The data must be stored in the int array in row-major order using the standard 24-bit color format (0xRRGGBB). The method parameters are `dest` - The mutable destination Image whose pixels will be set, and `rgbData` - The array of RGB pixel values.
- `public static Image getScaleImage(javax.microedition.lcdui.Image src, int width, int height, int method)` throws `IllegalArgumentException` – Creates a scaled version of the source image using the desired scaling method. All platforms must implement the SCALE_REPLICATE scaling method; other scaling methods may be optionally supported. SCALE_REPLICATE is used if the requested scaling method is not supported by the device. The method parameters are the following: `src` - the source Image; `width` - the width, in pixels, of the new Image, `height` - the height, in pixels, of the new Image, and `method` - The desired method to be used to scale the image data (see the item 0).

Using ImageUtil

This example uses an image (tank.png) to create a data structure (`rgbData`) to stores the RGB pixel data. The `rgbData` is used to draws the same image. Follows the example:

```
try {
     Image tank = Image.createImage("tank.png");
} catch(Exception e) {
  // The image can't be loaded
}

// creates a data structure to stores the RGB pixel data
from Image
int rgbData[] = new int[tank.getHeight()*tank.getWidth()];
// Stores the RGB pixel data from Image
ImageUtil.getPixels(tank,rgbData);
// Draws the image pixel by pixel with the respective RGB
pixel data
for (i=0;i<tank.getHeight();i++) {
     for (j=0;j<tank.getWidth();j++) {
           g.setColor(rgbData[i*tank.getWidth() + j]);
           g.fillRect(j,i,1,1);
     }
}
```

PalleteImage Class

`PaletteImage` provides methods for manipulating the color palette data of an image. **`PaletteImages`** can only be created with palette-based image data (PNG color type 3, or other palette image formats that a particular device may support).

The developer can retrieve either a single palette entry or the entire palette as a series of RGB values in 0xRRGGBB format (MIDP color format). The developer can also update a single entry or the entire palette by providing a new set of RGB values. The effects of the palette changes will be visible in the next Image that is generated.

Single color transparency is supported: the entire palette may be fully opaque, or a single palette entry may be designated as being fully transparent. Alpha channels are not supported.

Once the palette entries have been set to the desired values, a MIDP Image object is retrieved that reflects the new palette settings.

PalleteImage Constructor

The **`PalleteImage`** class defines the following constructors:

- `PalleteImage(byte[] data, int offset, int length)` throws IOException - Creates a new PalleteImage using the provided image data.
- `PalleteImage (java.lang.String name)` throws IOException - Creates a new PaletteImage using the provided image data in a named resource.

PalleteImage Methods

The **`PalleteImage`** class defines the following methods:

- `public Image getImage()` - Creates and returns a new Image object using this PaletteImage. The Image returned will reflect the PaletteImage's original pixel data and current palette data. This method enables the developer to easily generate a series of differently colored images by adjusting palette data.
- `public int getTransparentIndex()` – Gets the current transparent index. Pixels that reference the transparent index in the palette are not drawn when the image is rendered. By default, the transparent index is -1 even if a transparent color is specified in the original image data.
- `public void setTransparentIndex(int index)` throws `IndexOutOfBoundsException` – Sets the current transparent index. Pixels that reference the transparent index in the palette are not drawn when the image is rendered. The effects of the new transparent index will be reflected in the next Image object that is created by calling getImage().
- `public int getPaletteSize()` - Gets the number of entries in the palette.
- `public int getPaletteEntry(int index)` throws `IndexOutOfBoundsException` - Gets the specified entry in the palette. The method returns the current color value of the entry (0xRRGGBB format).
- `public void setPaletteEntry(int index, int color)` throws `IndexOutOfBoundsException` – Sets the specified entry in the palette. The color must be specified using the MIDP color format (0xRRGGBB, the upper 8 bits are ignored). The effects of the new palette will be reflected in the next Image object that is created by calling getImage().
- `public int[] getPalette()` – Gets the entire palette as an array of ints, each one representing a 24-bit RGB value. The method returns a new int array each time it is called, so this method should be used sparingly to avoid creating excessive garbage.
- `public void setPalette(int[] newPalette)` throws `ArrayIndexOutOfBoundsException, NullPointerException, IllegalArgumentException` - Sets the palette data for this image. The palette data must be specified using MIDP color format (0xRRGGBB, the upper 8 bits are

ignored). The size of the new palette must be at least as large as the value returned by `getPaletteSize()`; additional palette entries, if present, are ignored. The effects of the new palette will be reflected in the next Image object that is created by calling `getImage()`.

Using PalleteImage

`PalleteImage` enables a developer to adjust the colors of an image to match the capabilities of the device. It also enables reuse of image data by allowing the developer to change the color scheme. For example, a racing game may use a single **`PaletteImage`** of a car; the developer may then tweak the palette and generate a series of **`Image`**s of differently colored cars:

```
PaletteImage raceCar = new PaletteImage("car.png");

// Set the car color to red and retrieve the Image
raceCar.setPaletteEntry(0, 0xFF0000);
Image redRaceCar = raceCar.getImage();

// Set the car color to blue and retrieve the Image
raceCar.setPaletteEntry(0, 0x0000FF);
Image blueRaceCar = raceCar.getImage();

// Set the car color to green and retrieve the Image
raceCar.setPaletteEntry(0, 0x00FF00);
Image greenRaceCar = raceCar.getImage();

// The PaletteImage can now be discarded since we have the
// Image objects that we need
raceCar = null;
```

PlayField Class

A **`PlayField`** is a rectangular grid of cells with a set of available tiles to place in those cells and a set of associated Sprites.

The **`PlayField`** grid is made up of (rows * columns) cells, where the number of rows and columns are defined by parameters to the constructor. The cells are equally sized. The size of the cells is defined by the size of the tiles, or if the **`PlayField`** has no tiles, by arguments to the constructor. Each cell is either empty or contains a single tile whose image will be drawn in that cell. An empty cell is fully transparent - Nothing will be drawn in that area by the **`PlayField`**.

The tiles used to fill the **`PlayField`** cells can be either static tiles or animated tiles. Tiles are referred to using index numbers. Tile 0 (tile with index 0) refers to the special

empty tile. Any cell assigned the tile 0 will be considered empty and will be effectively transparent.

The static tile indices are non-negative (>=0) and the animated tiles indices are negative (<0).

Using Static and Animated Tiles

Static tiles are called static because their image does not often change, i.e., any cell that contains the static Tile 1 will always be drawn as the unchanging image of Tile 1. Tile 0 is a special static tile. It represents an empty cell. Any cell containing tile 0 will be transparent, it will not have a tile image drawn in it.

Animated tiles are called animated because their appearance changes easily over time. At any given time, each animated tile is associated with a particular static tile. When a cell containing an animated tile is drawn, the image of the static tile currently referenced by that animated tile will be drawn in that cell. In effect, the animated tiles provide indirect references to the set of static tiles, and therefore allow many cells to be animated simultaneously. For example, cells (0,0) and (0,1) both contain animated Tile -2. Animated Tile -2 currently references static Tile 1. Cells (0,0) and (0,1) will then be drawn with the image of static Tile 1. If animated Tile -2 is subsequently set to reference static Tile 2 by calling `setAnimatedTileImage(-2, 2);`, cells (0,0) and (0,1) will then be drawn with the image of static Tile 2.

Using Sprites

In addition to being a grid of cells, a **`Playfield`** can have a set of associated **`Sprites`** (see 0 item). When the **`PlayField`** is drawn, the grid is considered to have depth 0. Therefore, **`Sprites`** below the grid (**`Sprites`** with `Sprite.getDepth() < 0`) are drawn first. Then all cells in the grid are drawn. Then all the **`Sprites`** above the grid (`Sprite.getDepth() >= 0`) are drawn. The **`Sprites`** are drawn according to their location and visibility status as defined in the **`Sprite`** class. The location of **`Sprites`** is relative to the top-left corner of the **`PlayField`**.

Defining View Windows

A view window onto the **`PlayField`** can be defined using the method `setViewWindow()`. This defines the area of the **`PlayField`** that will be drawn by the `draw()` method. The default viewing window onto a **`PlayField`** (at construction time) is the entire area of the **`PlayField`**.

PlayField Constructor

The **`PlayField`** class defines the following constructors:

- PlayField (`int columns, int rows, Image img, int tWidth, int tHeight)` throws NullPointerException, IllegalArgumentException - Creates a new **PlayField** with a tile set. The parameter are the following:
  - `columns` - width of the PlayField in number of cells;
  - `rows` - height of the PlayField in number of cells;
  - `img` - Image to use for creating tiles;
  - `tWidth` - width, in pixels, of the individual tiles;
  - `tHeight` - height, in pixels, of the individual tiles.

  It creates a new **`PlayField`**, `rows` cells high and `columns` cells wide. The static tile set for the PlayField will be created from subsections of the image passed in. The **`PlayField`** grid is initially filled with empty cells (tile 0 - a reserved tile and represents an empty square) and laying out tiles must be accomplished through the use of other methods in the class. The creation of the static tile set follows these standards:

  Tiles must be equally sized, all being of the tile width (`tWidth`) and height (`tHeight`) defined in the constructor parameters. They may be laid out in the image horizontally, vertically, or as a grid. The width of the source image must be an integer multiple of the tile width. The height of the source image must be an integer multiple of the tile height.

  The tiles in the source image will have indices as follows:

  The static tiles are indexed like words are read on a page; left-to-right, then top-to-bottom. The top-left tile is assigned index 1. If there is a tile to its right, this tile is assigned index 2, and so on, across the first row of tiles. If there is a second row of tiles, the index of the left-most tile in this row is one greater than the right-most tile in the preceding row. Below is is a diagrammatic depiction:

1	2	. . .	N
N+1	N + 2	. . .	2N
2N+1	2N + 2	. . .	
. . .	. . .	. . .	. . .
[(M – 1) * N]+1	[(M-1) * N)] + 2	. . .	(M * N)

  So the total number of tiles is M * N, where:

  - N = (image width) / (tile width);
  - M = (image height) / (tile height).

  The indices for the static tile set will be non-negative (>=0) and the indices for animated tiles will be negative (<0). The index sets do not overlap and therefore

indices for static and animated tiles can be used interchangeably in the methods that set or move the contents of the **`PlayField`** cells. The static Tile set shall behave as if the image used in creation were cached. If a mutable image is used to create the tiles, the tiles' appearances should not reflect changes to the mutable source image. The appearance of individual static tiles can be changed with `setStaticTileImage()`. The entire static tile set can be changed using `setStaticTileSet()`. These methods should be used sparingly since they are both memory and time consuming.

- PlayField (`int columns, int rows, int cellWidth, int cellHeight)` throws IllegalArgumentException - Creates a new **PlayField** without a tile set. The parameter are the following:
  - `columns` - width of the PlayField in number of cells;
  - `rows` - height of the PlayField in number of cells;
  - `cellWidth` - Pixel width of each cell;
  - `cellHeight` - Pixel height of each cell.

  It creates a new **`PlayField`**, `rows` cells high and `columns` cells wide. A PlayField created with this constructor will not have any tiles (animated or static) associated with it. The primary use of a **`PlayField`** without tiles is expected to be as a container for managing sprites. The on-screen pixel dimensions of cells is defined by the parameters `cellWidth` and `cellHeight`. The cells in the PlayField are all empty (tile 0 - a reserved tile and represents an empty square). A tile set can later be added using `setStaticTileSet()`.

PlayField Methods

The **`PlayField`** class defines the following methods:

- `public void addSprite(Sprite s)` throws `NullPointerException` – Add a Sprite to the PlayField. Ignores the request if the Sprite is already associated with the PlayField.
- `public void removeSprite(Sprite s)` throws `RuntimeException`, NullPointerException – Remove a Sprite from PlayField.
- `public void removeAllSprites()` – Remove all Sprites from PlayField.
- `public int createAnimatedTile(int staticTileIdx)` throws `IndexOutOfBoundsException` – Creates a new animated tile and initializes it with a static tile index. Returns the index to use when referring to this animated tile. The indices for animated tiles will be negative (<0) and the indices for the static tile set will be positive (>=0). The index sets do not overlap and therefore indices for static and animated tiles can be used interchangeably in the methods that set or move the contents of the PlayField cells. The first animated tile shall have the index -1, the second, -2, etc.

- `public void setAnimatedTileImage(int animTileIdx, int staticTileIdx)` throws `IndexOutOfBoundsException` – Sets the static tile that will be displayed in any cell that contains the animated tile. The method parameters are `animTileIdx` - index of the animated tile, and `staticTileIdx` - index of a static tile to be referenced by the animated tile.
- `public int getAnimatedTileImage(int animTileIdx)` throws `IndexOutOfBoundsException` – Get the static tile referenced by an animated tile, and returns the index of the static tile that is currently referenced by an animated tile.
- `public void setCell(int celCol, int celRow, int tileIdx)` throws `IndexOutOfBoundsException`, `ArrayIndexOutOfBoundsException` – Sets the tile to be displayed in a cell. The tile can be either a static or an animated tile. The method parameters are `celCol` - column of cell to set, `celRow` - row of cell to set, and `tileIdx` - index of tile to place in cell.
- `public int getCell(int celCol, int celRow)` throws.`ArrayIndexOutOfBoundsException` – Gets the index of the static or animated tile currently displayed in a cell.
- `public void moveTiles(int dstCol, int dstRow, int srcCol, int srcRow, int width, int height)` throws `ArrayIndexOutOfBoundsException` – Move a rectangular set of tiles from a source location to a destination location. Source cells are left empty. If the source and destination cells overlap, the method shall behave as if the source cells are first copied to a separate array, the source cells are cleared, and the tiles are then copied back to the destination cells. The method parameters are the following: `dstCol` - column of top-left destination cell; `dstRow` - row of top-left destination cell; `srcCol` - column of top-left source cell; `srcRow` - row of top-left source cell; `width` - width, in rows, of area of tiles to move; and `height` - height, in rows, of area of tiles to move.
- `public void fillCells(int col, int row, int width, int height, int tileIdx)` throws `IndexOutOfBoundsException`, `ArrayIndexOutOfBoundsException` – Fill each cell in a rectangular area with a given animated or static tile. The method parameters are the following: `col` - column of top-left cell; `row` - row of top-left cell; `width` - width, in rows, of area of cells to fill; `height` - height, in rows, of area of cells to fill; and `tileIdx` - index of tile to place in fill region.
- `public void draw(javax.microedition.lcdui.Graphics g, int x, int y)` throws `NullPointerException` – Draw the PlayField to a Graphics instance, anchoring the top left corner of the PlayField view window at the position `(x, y)` on the Graphics instance. The method parameters are the following: `g` - Graphics instance on which to draw the PlayField; `x` - the x coordinate of the top left corner of the PlayField; and `y` - the y coordinate of the top left corner of the PlayField. The PlayField will be drawn as follows:

  - Draw all the `Sprites` with depth < 0 in increasing order of depth (depth -2 drawn before or below depth -1);
  - Draw the tiles for all cells. Empty cells, those with Tile 0, are considered fully transparent, so nothing is drawn for them.
  - Draw the all the `Sprites` with depth >= 0 in increasing order of depth (depth 1 drawn before or below depth 2). The location of the Sprites is defined by the `Sprite` instance and is relative to the top left corner of the `PlayField` grid.

- `public int getCellWidth()` – Get width of a cell, in pixels.
- `public int getCellHeight()` – Get height of a cell, in pixels.
- `public int getGridWidth()` – Get width of the PlayField grid, in cells.
- `public int getGridHeight()` – Get height of the PlayField grid, in cells.
- `public boolean anyCollisions()` – This method checks whether any of the PlayField's Sprites collide with any of the PlayField's tiles or other Sprites on the PlayField. It will return `true` if any Sprite on the PlayField collides with a tile or any other Sprite. Like collidesWithSprites(Sprite) and collidesWithAnyTile(Sprite), this method reports collisions only at a boundary level granularity, not pixel level granularity.
- `public boolean collidesWithSprites(Sprite s)` throws `NullPointerException` – Check for Sprite collision with any other Sprites on the PlayField. This method is complemented by Sprite.collidesWith(Sprite, boolean); If collidesWithSprites(Sprite) returns true, the developer can find the exact Sprite collision(s) by using Sprite.collidesWith(Sprite, boolean). This is similar to how collidesWithAnyTile(Sprite) and collidesWithTiles(int, int, int, int, Sprite, boolean) complement each other. Sprite `s` does not have to have been added to the PlayField. The collision detection will proceed as if the Sprite is on the PlayField. That is, its location will be treated as relative to the origin of the PlayField's coordinate system.
- `public boolean collidesWithAnyTile(Sprite s)` throws `NullPointerException` – Check for Sprite collision with PlayField tiles. Return true if the Sprite overlaps with a cell that contains a tile (i.e. a cell containing a non-zero tile index). Sprite `s` does not have to have been added to the PlayField. The collision detection will proceed as if the Sprite is on the PlayField. That is, its location will be treated as relative to the origin of the PlayField's coordinate system.
- `public boolean collidesWithTiles(int col, int row, int width, int height, Sprite s, boolean pixelLevel)` throws `NullPointerException`, `ArrayIndexOutOfBoundsException` – Check for Sprite collision with a region of PlayField tiles. It returns true if the Sprite overlaps with a cell in the defined region that contains a tile (i.e. a cell containing a non-zero tile index). If `pixelLevel` is true, this method will report a collision only when opaque Sprite pixels overlap opaque tile pixels. This method complements the collidesWithAnyTile(Sprite) method by letting the programmer focus their search and find specific tiles or regions of collision. This is similar to how Sprite.collidesWith(Sprite, boolean) complements collidesWithSprites(Sprite). The method parameters are the following: `row` - Row of top-left cell for collision check

region; `col` - Column of top-left cell for collision check region; `height` - Height, in rows, of area for collision check; `width` - Width, in rows, of area for collision check; `s` - Sprite to check for collision; and `pixelLevel` - Boolean indicating whether collision detection should be done at a pixel level instead of simply as boundary checks.

- `public void setStaticTileImage(int staticTileIdx, Image img, int x, int y)` throws `NullPointerException`, `ArrayIndexOutOfBoundsException` – Modify the image associated with a static tile. Replace the image currently associated with a static tile with a new image of the same size. New static tile image will be extracted from the image passed in, starting from pixel `(x, y)` in the new source image and extending for getCellWidth() pixels horizontally and getCellHeight() pixels vertically. As at tile set creation time, if a mutable source image is used, behavior of the tile set should be as if the new image were cached. Updates to the mutable source image will not cause a change in the appearance of the tile image.
- `public void setStaticTileSet(Image img, int tWidth, int tHeight)` throws `NullPointerException`, `IllegalArgumentException` – Replaces the current static tile set with a new static tile set. See the constructor PlayField(int, int, Image, int, int) for information on how the tiles are created from the image. If the new static tiles have the same dimensions as the previous static tiles, the view window will be unchanged. If the new static tiles have different dimensions than the previous static tiles, the view window will be reset to the construction default, i.e. the entire grid dimension. If the new static tile set has as many or more tiles than the previous static tile set, then the animated tiles will be unchanged, and the contents of the PlayField grid will be unchanged. If the new static tile set has less tiles than the previous static tile set, then the PlayField grid will be reset to completely empty, and All animated tiles will be deleted.
- `public void setViewWindow(int x, int y, int width, int height)` – Sets the portion of the PlayField that will be drawn when draw(Graphics, int, int) is called. This will limit the portion of the PlayField that is drawn to the rectangle defined by the region `(x, y)` to `(x + width, y + height)`. The default view window (at construction time) is the entire area of the PlayField, i.e. the rectangular region bounded by `(0, 0)` and (getGridWidth() * getCellWidth(), getGridHeight() * `getCellHeight()`).The rectangle defined by the parameters may extend beyond the bounds of the PlayField. If this happens, the `draw(graphics, int, int)` method will draw no tiles in the area outside the grid boundaries. Sprites may still be drawn in this area if their position places them outside the bounds of the PlayField grid. The view window stays in effect until it is modified by another call to this method or is reset as a result of calling setStaticTileSet(Image, int, int). The method parameters are `x` - x coord of top-left pixel for the drawing view window, `y` - y coord of top-left pixel for the drawing view window, `width` - width of the drawing view window, and `height` - height of the drawing view window.

Using PlayField

Follows a **`PlayField`** example:

```
// Creates a playField with 100 columns and 10
// rows and tiles with 24x16 pixels
PlayField foreground = new PlayField(100, 10,
Image.createImage("tiles.png"), 24, 16);
// Sets the first cell in the first line to
// empty(tile with index 0)
foreground.setCell(0, 0, 0);
// Fills the second cell in the first line with tile 1
foreground.setCell(1, 0, 1);
// Fills the third cell in the first line with tile 2
foreground.setCell(2, 0, 2);
// Fills the fourth cell in the first line with tile 3
foreground.setCell(3, 0, 3);
// Gets the Graphics object for this GameScreen
Graphics g = getGraphics();
// Draws the foreground playfield
foreground.draw(g, 0, 0);
```

SoundEffect Class

The **`SoundEffect`** class encapsulates the data for a game sound effect. A game may create several **`SoundEffect`** objects, one for each of the sounds that it needs to play. The sound data may be stored on the device as a named resource in the application JAR file, or it can be stored on a server and retrieved via the network. **`SoundEffect`** instances are played by a **`GameScreen`**.

SoundEffect Methods

The **`SoundEffect`** class implements the following method:

- `public static SoundEffect createSoundEffect(String resource)` throws `FileFormatNotSupportedException` – Creates a **SoundEffect** for the sound data stored in the specified named resource or URL. The data must be in a sound format that is supported by the device. Though additional formats may be supported, all devices must support some format yet to be determined.

Using SoundEffect

As described above, a game can need several different sound effects. The code below exemplifies the creation of some **`SoundEffect`** objects:

```
try{
   // Create a SoundEffect using a wave file inside the JAR
   SoundEffect s1 = createSoundEffect("/jump.wav");

   // Create a SoundEffect using a wave located
   // on a web site
   SoundEffect s2=

createSoundEffect("http://www.motorola.com/sound/mp.wav");

}catch(FileFormatNotSupportedException fe){}
```

Sprite Class

The **`Sprite`** class is used to create graphic images, animated or non-animated, that a user can interact with and move around.

Animation Frames

An animated **`sprite`** is created from an image divided into sections as described in the constructor Sprite(Image, int, int). The individual sections of the image are considered the raw frames of the **`Sprite`**. The method getNumRawFrames returns the number of raw frames.

Sprite Drawing

`Sprites` can be drawn at anytime using the `draw(Graphics)` method. The sprite will be drawn on the **`Graphics`** object, according to the current state information maintained by the Sprite (i.e. position, frame, visibility). Some potential uses of Sprites include:

- Arbitrarily draw the Sprite on a **GameScreen**.
- A Sprite can be added to a **PlayField**. Then PlayField.draw(Graphics, int, int) will automatically draw all the Sprites associated with the **PlayField**.
- draw(Graphics) could be called from the `paint()` method in a subclass of **Canvas**.
- draw(Graphics) could be called at any time to draw the **Sprite** on a MIDP mutable image. This is virtually identical to the first bullet, drawing on a **GameScreen**.

Only in the case where a set of Sprites are a part of a container object (i.e. where the **Sprite** is associated with a **PlayField**) is the depth information automatically handled by the system. In other situations, managing the drawing order is the responsibility of the developer.

Sprite Constructor

The **`Sprite`** class defines the following constructors:

- `public Sprite(Image img)` – Creates a new non-animated **Sprite** from an **Image** object. All animation operations on a non-animated Sprite behave as if there is a single raw frame. At construction time, the **Sprite**'s position will be set to (0,0), the depth will be set to 0, and the Sprite will be visible. The Sprite shall behave as if the image used in creation were cached. If a mutable image is used to create the Sprite, the Sprite's appearance should not reflect changes to mutable source image.
- `public Sprite(Image img, int fWidth, int fHeight)` – Creates a new animated **Sprite** from an Image. The constructor parameters are the following:
  - img - Image to use for Sprite;
  - fWidth - width, in pixels, of the individual raw frames;
  - fHeight - height, in pixels, of the individual raw frames.

  The creation of the raw frames follows these standards:
  - Frames must be equally sized, all being of the frame width (`fWidth`) and height (`fHeight`) defined in the constructor parameters. They may be laid out in the image horizontally, vertically, or as a grid. The width of the source image must be an integer multiple of the frame width. The height of the source image must be an integer multiple of the frame height.

The frames in the source image will have raw frame numbers as follows:

- The frames are numbered like words are read on a page; left-to-right, then top-to-bottom. The top-left frame is numbered 0. If there is a frame to its right, this frame is numbered 1, and so on, across the first row of frames. If there is a second row of frames, the number of the left-most frame in this row is one greater than the right-most frame in the preceding row. Below is a diagrammatic depiction:

0	1	. . .	N – 1
N	N + 1	. . .	2N – 1
2N	2N + 1	. . .	
. . .	. . .	. . .	. . .
(M – 1) * N	((M-1)* N) + 1	. . .	(M * N) – 1

So the total number of frames is M * N, where:

- N = (image width) / (frame width)
- M = (image height) / (frame height)

At the time of creation, all Sprites have a default frame sequence corresponding to the raw frame numbers. This can be modified with `setFrameSequence()`. At construction time, the Sprite's position will be set to (0,0), the depth will be set to 0, and the Sprite will be visible. The Sprite shall behave as if the image used in creation were cached. If a mutable image is used to create the Sprite, the Sprite's appearance should not reflect changes to mutable source image.

- `public Sprite(Sprite s)` – Creates a new Sprite from another **Sprite**. Create a copy of a **Sprite**. All attributes (raw frames, position, frame sequence, current frame, visibility) of the source **Sprite** should be reflected in the new **Sprite**. Any subsequent updates to the source **Sprite** after the creation of the second Sprite should *not* be reflected in the second Sprite.

Sprite Methods

The **Sprite** class implements the following methods:

- `public void setPosition(int x, int y)` – Set Sprite's x,y position. The x, y position is relative to whatever object the sprite is associated with or drawn on.
- `public void setDepth(int d)` – Set Sprite's depth order. The depth order is relative to other Sprites when multiple Sprites are contained in a container object, i.e. a PlayField. When Sprites are drawn explicitly instead of implicitly through the use of a container object, the management of drawing order is the responsibility of the developer. Integer.MIN_VALUE is the lowest depth, Integer.MAX_VALUE is the highest depth. So items with depth Integer.MIN_VALUE would be drawn first, or at the bottom, and items with depth Integer.MAX_VALUE would be drawn last or on top.
- `public void move(int dx, int dy)` – Move Sprite. The method parameters are `dx` - pixels to move Sprite along horizontal axis, and `dy` - pixels to move Sprite along vertical axis.
- `public int getX()` – Get Sprite's x position.
- `public int getY()` – Get Sprite's y position.
- `public int getDepth()` – Get Sprite's depth order.
- `public int getHeight()` – Get Sprite's height order.
- `public int getWidth()` – Get Sprite's width in pixels.
- `public boolean collidesWith(Sprite s, boolean pixelLevel)` throws `NullPointerException` – Check for collision between two Sprites. If `pixelLevel` is false, check for overlap in the rectangular areas of the two Sprites, using positions (x, y) and extents (width, height).

The two Sprites are treated as if they are in the same coordinate system. For example, if the two Sprites are on different PlayFields that are drawn at different locations, this method still behaves as if they are on the same PlayField. If `pixelLevel` is true, check for overlap in opaque pixels of the two Sprites. Overlapping in transparent regions of either Sprite will not be considered a collision.

- `public void setFrame(int frame)` – Set Sprite's animation frame. Sets which frame from the frame sequence to draw when draw(Graphics) is called. All Sprites have a default frame sequence as described in the constructor.
- `public int getFrame()` – Get Sprite's current animation frame. All Sprites have a default frame sequence as described in the constructor.
- `public int getNumRawFrames()` – Get the number of raw frames in the original frame set for this Sprite.
- `public void nextFrame()` – Set current animation frame to the next frame. Advance to next frame in the frame sequence. All Sprites have a default frame sequence as described in the constructor. Frame list is considered to be circular, i.e. if `nextFrame()` is called when the last frame is the current frame, this will advance to the first frame.
- `public void prevFrame()` – Set current animation frame to the previous frame. Advance to previous frame in the frame sequence. All Sprites have a default frame sequence as described in the constructor. Frame list is considered to be circular, i.e. if `prevFrame()` is called when the first frame is the current frame, this will advance to the last frame.
- `public void setVisible(boolean visible)` – Set visibility status. If `setVisible(false)` is called, the Sprite will not be drawn by `draw(Graphics)` until `setVisible(true)` is called.
- `public boolean isVisible()` – Get visibility status. The method returns boolean indicating whether the Sprite will be drawn by `draw(Graphics)`.
- `public final void draw(Graphics g)` throws `NullPointerException` – Draw the Sprite. Draw current frame of Sprite to Graphics instance g at location currently set in Sprite. Sprite will be drawn only if `isVisible()= true`.
- `public void setFrameSequence(int[] seq)` throws `ArrayIndexOutOfBoundsException` – Set the sequence of frames to cycle through with next/prevFrame. All Sprites have a default sequence as described in the constructor. This method allows for the creation of an arbitrary sequence from the original frameset. The methods nextFrame(), prevFrame(), getFrame(), and setFrame(int) all operate on the frame sequence. Passing in `null` causes the sequence to revert to the default sequence defined in the constructor. The parameter `seq` is an array of integers, where each integer is a reference to a frame in the original raw frameset, that is, the frames from left to right on the original image.
- `public int[] getFrameSequence()` – Get the current frame sequence. Returns the frame sequence set with `setFrameSequence(int[])` or, if none has been set, return the default

frame sequence for this Sprite. Each entry in the array is an index to the original raw frameset, that is, the frame numbering as described in the constructor.

- `public void setImage(javax.microedition.lcdui.Image img, int fWidth, int fHeight)` throws `NullPointerException`, `IllegalArgumentException` – Change the image used for the Sprite. Replaces the current raw frames of the Sprite with a new set of raw frames. See the constructor Sprite(Image, int, int) for information on how the frames are created from the image. Changing the image for the Sprite could change the number of raw frames. If the new frame set has as many or more raw frames than the previous frame set, then:
  - The current frame will be unchanged;
  - If a custom frame sequence has been defined (using setFrameSequence(int[])), it will remain unchanged. If no custom frame sequence is defined (i.e. the default frame sequence is in use), the default frame sequence will be updated to be the default frame sequence for the new frame set. In other words, the new default frame sequence will include all of the frames from the new raw frame set, as if this new image had been used in the constructor.

  If the new frame set have less frames than the previous frame set, then:

  - The current frame will be reset to frame 0;
  - Any custom frame sequence will be deleted and the frame sequence will revert to the default frame sequence for the new frame set (all frames in the frame set, left-to-right then top-to-bottom).

Using Sprite

The example below creates two Sprites (bullet and tank) and tests collisions between them. When there are no lives left, the game finishes.

```
try {
    Sprite bullet = new
Sprite(Image.createImage("bullet.png");
    Sprite tank = new Sprite(Image.createImage("tank.png");
} catch (Exception e) {
// any image can't be loaded
}
Boolean isGameOver= False;
int lifes= 3; // The number of lives is 3
while(!isGameOver) {
   // verifies the collision between the two sprites
   if(tank.collidesWith(bullet,false)) {
      lifes--;
      // If there are no more lifes, the game is over
      if(lifes == -1) {
          isGameOver = true;
```

```
            }
        }
    }
```

FileFormatNotSupportedException

The **`FileFormatNotSupportedException`** is an exception which will be thrown when a **`SoundEffect`** or **`BackgroundMusic`** format is not supported by the platform or the size of the data is larger than the size of the internal buffers. The **`FileFormatNotSupportedException`** extends the **`java`**.**`lang`**.**`RuntimeException`** class.

FileFormatNotSupportedException Constructors

The **`FileFormatNotSupportedException`** class defines the following constructors:

- `public FileFormatNotSupportedException( java.lang.Exception e)` – The parameter `e` is the underlying exception that caused the failure.
- `public FileFormatNotSupportedException( java.lang.String info)` – The parameter `info` is a String containing information about the failure
- `public FileFormatNotSupportedException( java.lang.String info, Exception e)` – The parameters are `info` a String containing information about the failure, and `e` - The underlying exception that caused the failure.

Appendix A: Key Mapping of Motorola A830 handset

Key Mapping

The Figure 19 maps out the keys available through the `javax.microedition.lcdui.Canvas` class. By overriding the `Canvas.keyPressed()` and `Canvas.keyReleased()` methods, the MIDlet can listen for certain key presses and key releases. The keys available to the MIDlet via the MIDP specs include the ITU-T keypad (0-9, *, #). In addition to the standard keys, the Motorola A830 handset offers a 4-way keypad, two soft keys, a menu key, and a send key

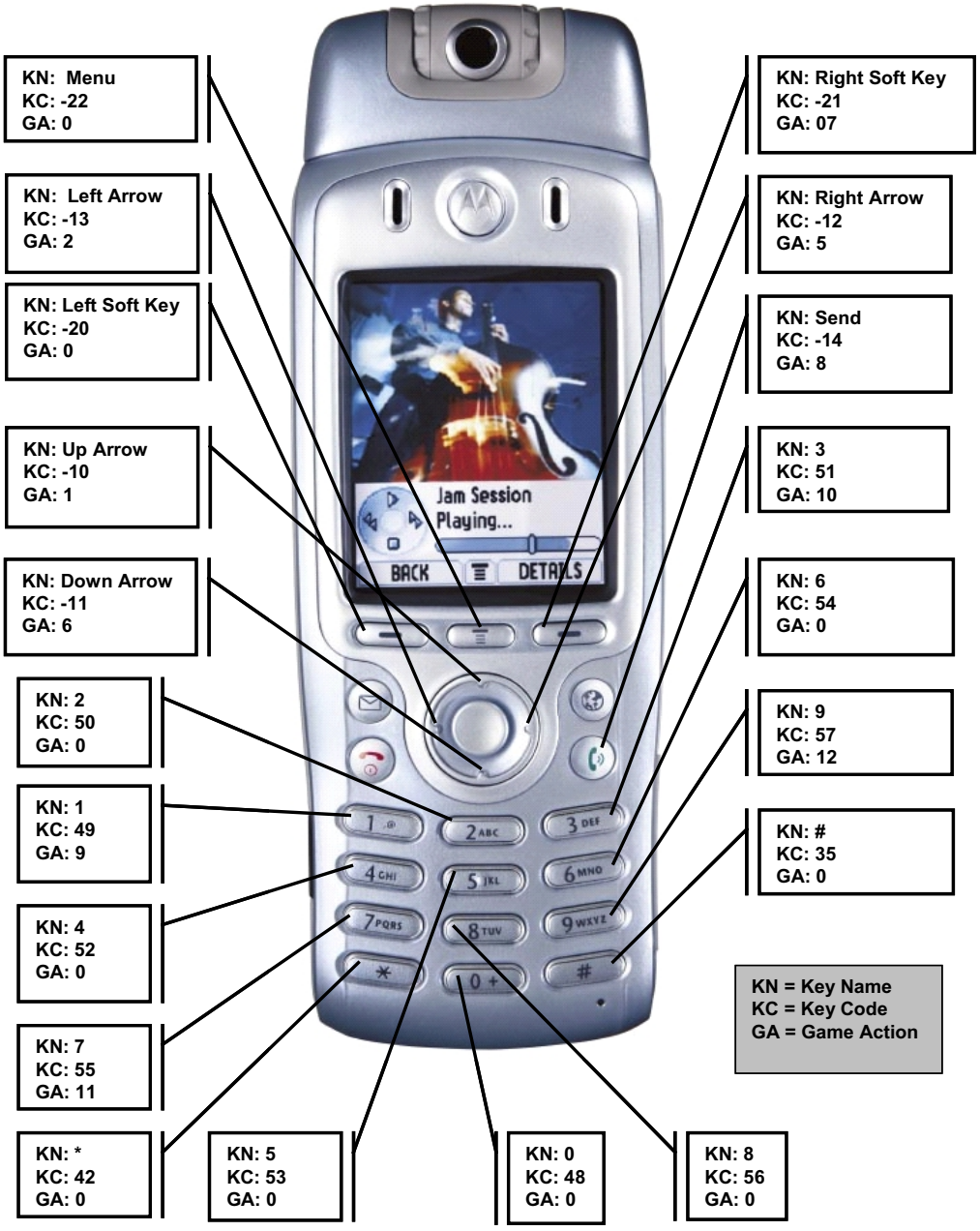

Figure 19. Key Code Mapping

Appendix B: How To

Downloading to the Device

Serial port download procedure

The MIDway utility provides the application developer a means to load an application to the A830 device through PC. A serial data cable is used to connect to the bottom connector on the Motorola A830 handset, and the PC serial port.

Figure 20. Serial port download

For instructions on installing the MIDway utility, please consult the user guide.

After loading the JAR and JAD file on the Motorola A830 handset, the friendly name specified in the MANIFEST.MF file for the MIDlets should appear on the Games & Apps menu. For the HelloWorld example, the Games & Apps menu will contain an item "HelloWorld" representing the application. At this point, the application is only "Loaded" on the Motorola A830 handset and not yet installed. From this point, the application may be removed from the device.

NOTES: The number of MIDlet suites that can be installed on the Motorola A830 handset is limited to 20. If the number of MIDlets suites installed is more than 20, de-install an application before proceeding.

The application must be installed before it can be executed. The following steps describe the installation procedure.

OTA procedure

There is no need of additional software tools, or cables, to downloaded MIDlet suites through WAP browser. The application developer just has to use the A830 browser and connect to a WAP server site that contains the desired MIDlet suite to be downloaded. The application developer should follow the WAP server site instructions to download the MIDlet properly.

After loading the JAR and JAD file on the Motorola A830 handset, the friendly name specified in the MANIFEST.MF file for the MIDlets should appear on the Games & Apps menu.

Installation

The following checklist should be covered before attempting to install a MIDlet Suite. Failure to verify this checklist could lead to an installation failure.

- Applications supports CLDC-1.0 and MIDP-1.0 (the configuration and profile supported by the Motorola A830 handset)
- JAD file has been created.
- JAR file contains META-INF/MANIFEST.MF.
- Verify the MIDlet-Name, MIDlet-Version, and MIDlet-Vendor attributes are duplicated in both the MANIFEST.MF and the JAD file.
- Both the JAD and JAR file have the same name (except for the .JAD and .JAR extensions).
- File names (JAR and JAD) are less than 32 characters (not including extension).
- Less than 20 MIDlet suites are currently installed.
- Maximum length of class path inside JAR file must be 64 characters.
- Maximum length of URL path must be 256 characters.
- No more than ~500 files are used by installed MIDlet suites.
- JAR size listed in JAD matches actual JAR size.
- MIDlet suite version must be higher than an already installed one.

Even though the Data and Program Space in Java System indicate more available space than the size of a particular JAR file, it doesn't necessarily mean the JAR will install.

Moreover, if it is able to install, there's no guarantee the MIDlet will execute because quite often more RAM is required for execution and then installation. In addition, MIDlets that will not install or execute on the phone because of lack of memory will most certainly execute on the Sun Wireless Toolkit since the PC has virtually unlimited memory with respect to the size of MIDlets.

The memory requirements for MIDlet suite installation are the following:

- First, there must be enough Data Space (file system space) to temporarily store the JAR. If there's not enough Data Space, the browser (in the OTA mechanism) will display the error "Insufficient Memory".
- Secondly, there must be enough heap memory to uncompress the JAR file. The JAR size should be a predefined safe proportion of the heap size. The JAR maximum size recommended is 100K. This means that MIDlet typically will not install if the JAR is greater than 100K. There are exceptions to this and it depends on how many class files vs. resource files are contained within the JAR. If there's not enough heap, the device will typically display the message "Memory Full".
- Third, there must be enough Data Space to store not only the temporary JAR but also all the resource files needed by the MIDlet. The JAR is essentially a zip file that must be uncompressed. It contains class files (the actual application) and resource files that are used by the MIDlet. These resources typically include, png images, database files and any other data the MIDlet needs. These resource files are stored in the Data Space during installation. The JAR is deleted after the installation phase completes. If there's not enough Data Space, the device will typically display the message "Memory Full". Also, note that total size of the uncompressed resources in the JAR doesn't necessarily equal the Data Space occupied by that MIDlet once installed.
- Fourth, there needs to be enough Program Space to store the actual MIDlet. The class files in the JAR are the application files and are converted into a native format and stored in the Program Space during installation. This native format size will be greater than the total of the uncompressed class files in the JAR. Once stored in the Program Space, the MIDlets are referred to as DAV Objects. DAV reserves additional Program Space equal to the largest DAV Object. This reserved space cannot be used for additional MIDlets. Its purpose is to provide power loss protection during a DAV reclaim of the flash memory. The allocation of this reserved Program Space is often a point of confusion with users. When the largest DAV object is installed, the Program Space in Java System will be reduced by more than the size of Program Space in Suite Details. Java System shows the free Program Space. Suite Details shows the amount of Program Space occupied by that MIDlet.

Program and Data space notes:

- To check Program and Data space from the Java menu, select "Java System" and press the "Select" soft key.
- Program space is used to store class files.
- Data space is used to store the JAR files before installation and resource files after installation. After installation, the JAR file is destroyed.

Then to install the MIDlet Suite, highlight the Suite in the Java Tools menu. Select the "Java Application Loader" option and press the SELECT soft key. A dialogue will be displayed indicating the serial cable must be connected to the device. Execute the

MIDway tool on PC, select and send the desired MIDlet to be installed. The MIDway tool indicates exactly which steps are being executed.

Java Application Installer/De-Installer (JAID)

- JAID is a component built into the Motorola KVM to handle installation and de-installation of Java applications to a device. The process of installing an application is time intensive involving loading of the class files from the JAR file and writing the image, in a platform-specific manner, to memory. By installing Java applications, class files do not have to be stored in RAM, allowing more runtime memory for the application at hand. Additionally, the time required to launch Java applications is decreased dramatically.
- After successful installation, the class files are placed in the Program space and the resource files are placed in the Data space. The original JAR file is then destroyed.
- Applications only need to be JAID installed once. If the Motorola A830 handset's software is upgraded, Java applications must be re-installed.

Once the application is done installing on the Motorola A830 handset, you need to return to the Games & Apps menu to launch the downloaded application.

If you leave the installation progress screen while the MIDlet Suite is still being installed, the installation will fail, and you must repeat all installation procedure again.

Starting Applications

Often times a MIDlet Suite only contains one MIDlet. If so, then that MIDlet can be launched from the Games & Apps menu simply by highlighting that MIDlet Suite and pressing the "SELECT" soft key.

If there are multiple MIDlets in the Suite, then a suite content menu will be displayed, and one of the individual MIDlets can be highlighted. From there, pressing the "RUN" soft key will launch the selected MIDlet.

Exiting Applications

During the development process, chances are a MIDlet may not exit properly via the "correct" and "elegant" method. The Motorola A830 handset's policy on Java applications is to allow the user to exit an application at anytime, either forcefully or via a menu option. If an application, during the development process, becomes unstable or fails to respond, the user/developer may end the application by pressing the END key.

Appendix C: Frequently Asked Questions

Question 1 How is port configurations handled? What if native functionality is using a port?

Answer Ports used by native applications will be allocated first. If a MIDlet requests a port in use by a native application an IOException will be thrown. For other applications, use random port assignments.

Question 2 Are volume keys available to be mapped by the application? Which keys are going to be mapped and which ones are not?

Answer The following keys are not available. Everything else can be mapped by the application.

- End/Home

- Power

Question 3 What events will cause the application to turn over control to the native OS?

Answer The application never turns over control to the native OS because the native OS is always in control. The native OS will pause a MIDlet if the device receives an incoming Group Call, Private Call, or Phone Call.

Question 4 Will applications have a dedicated icon?

Answer The manifest has space for the user to specify a PNG image to use as an icon, however the Motorola A830 handset does not support MIDlet icons in the Java menu.

Question 5 Is there a “Sleep” mode?

Answer There is no “Sleep” mode. MIDlets can be paused therefore, application developers should write MIDlets in such a way that they become less active when paused: Longer delays in loops, etc.

Question 6 In addition to the CLDC 1.0 and MIDP 1.0 libraries that will come on the A830, what other libraries will be provided?

Answer The available APIs are LWT, KJava Telephony and Gaming.

Question 7 Does your Java implementation support HTTP as a communication protocol? If not, what else do you support?

Answer We support HTTP, HTTPS, TCP Sockets, UDP socket and serial port.

Question 8 What is the operating system of the phones? Which virtual machine do you use? Is it the same for all Motorola phones?

Answer The OS is an in-house proprietary RTOS. The KVM is licensed from Sun with in-house enhancements done by Motorola. All Motorola phones have the same KVM.

Question 9 Is it correct there would be a 100k limit to the size of the .jar files that can be downloaded to the phone? Is that a limit created by Motorola in the J2ME implementation or is it an overall MIDP limitation that I missed in the spec?

Answer 100k limit in terms of compressed JAR file size is a guideline. The limitation is due to limited memory in the phone and not specified by MIDP.

Question 10 If the application JAR size is limited to 100k, for example, can the JAR file be broken into several JAR files?

Answer No; you can download files bigger than 100k but it may fail to be installed. For the sake of performance, the Motorola A830 handset does not execute directly from the JAR. In-device loading and linking of class files to the VM is done before the application is executed the first time.

Question 11 How much RAM will be available? Total memory per application and stack?

Answer 512k heap is allocated. Only one MIDlet can run at a time, sharing a small portion of the heap with the KVM.

Question 12 What are the storage limitations of the phone?

Answer Total flash data space on the Motorola A830 handset is 1.2Mb, however this space is shared with other phone features like ring tones and uninstalled MIDlets. The amount available to a J2ME MIDlet will vary accordingly.

Question 13 What is the bandwidth available for http?

Answer The http bandwidth depends on the carrier and the link. Using AT&T GPRS the measured bandwidth is 3800 bytes/sec. It is possible that the bandwidth with UMTS is larger than 3800 bytes/sec.

Question 14 Do you have an access / security model in place for record management system?

Answer The unit of security domain is MIDlet suite. MIDlet suites are protected from each other. A MIDlet suite must be packaged in one JAR file.

Question 15 Where can I download the Motorola SDK for J2ME development?

Answer You can download the Motorola SDK for J2ME from:
`www.motorola.com/develoeprs/wireless/`

Question 16 Is there is any way to initiate a call from a J2ME application running on Motorola A830 handset?

Answer You cannot initiate a call from a J2ME application.

Question 17 Are there any J2ME libraries/methods where I can manipulate files or save files?

Answer The only methods available to manipulate files and save data use RMS package.
A recordstore consists of a collection of records, which will remain persistent across multiple invocations of the MIDlet. The platform is responsible for making its best effort to maintain the integrity of the MIDlet's record stores throughout the normal use of the platform, including reboots, battery changes, etc.
Recordstores are created in platform-dependent locations, which are not exposed to the MIDlets. The naming space for record stores is controlled at the MIDlet suite granularity. MIDlets within a MIDlet suite are allowed to create multiple record stores, as long as they are each given different names. When a MIDlet suite is removed from a platform installed recordstores associated with its MIDlets will also be removed. These APIs only allow the manipulation of the MIDlet suite's own record stores, and do not provide any mechanism for record sharing between MIDlets in different MIDlet suites. MIDlets within a MIDlet suite can access each other's record stores directly.

Question 18 Why do you need to handle IOExceptions and when can they be caused?

Answer When the network isn't available due to temporary problems or invalid settings on your phone, you get "PPP up failed."
When the network doesn't provide your phone with a DNS server or the DNS server can't be reached, you get "Can't obtain IP address."
When something is wrong with the response from a server or the way you handle the response, you get "malformed response." (need to work out exactly which cases give you a malformed response message).

Appendix D: Sun Microsystem's J2ME™ Wireless Toolkit

Overview

The J2ME Wireless Toolkit is a set of tools that provides developers with the emulation environment, documentation and examples needed to develop CLDC/MIDP compliant applications. To obtain detailed information such as the system requirements, installing and downloading the Wireless Toolkit, please refer to the J2ME Wireless Toolkit homepage .

One of the benefits of using the Wireless Toolkit is its flexibility to emulate any new platform such as the Motorola A830 handset. To customize the Wireless Toolkit for the Motorola A830 handset, Motorola provides the following items:

- Stubbed out A830 OEM APIs – Used as external class libraries
- Motorola A830 handset images – Skins for Motorola A830 handset
- Motorola A830 handset device property file – Device specific information
- JavaDocs for A830 OEM APIs

After install, to learn how to use the J2ME Wireless Toolkit, please refer to the installed directory: `{Installed dir}\docs\UserGuide.pdf`.

Here are some directories that developers should be aware of:

URL	Description
`lib\midpapi.zip`	Archive containing the CLDC and MIDP API classes. These files are used during the compilation of the application source files and the byte-code pre-verification of the application classes.
`apps\lib`	Contains external class libraries, in JAR or zip format. All MIDlets in the apps directory have access to these external class libraries.
`apps\{project`	Contains external class libraries, in JAR or zip format. Only

URL	Description
name}\lib	{project name} MIDlet has access to these external class libraries.

Customizing the Wireless Toolkit to the Motorola A830 handset

Use {installed dir}\docs\BasicCustomizationGuide.pdf to learn more about how to create a new device in the Wireless Toolkit.

As *BasicCustomizationGuide.pdf* mentions, there are only three steps to create a new device in the Wireless Toolkit:

1. Obtain the default J2ME Wireless Toolkit.

   The toolkit includes a default development environment and a Default Emulator. The Default Emulator is supplied with sets of device property files that enable the emulation of several generic wireless devices.

2. Create new device property files.

   A company that wants to have applications developed for a specific device using the toolkit can modify the device property file and use them with the Default Emulator. Download A830.zip from www.motorola.com/developes/wireless/.

3. Add the new device property files to the J2ME Wireless Toolkit.

   A set of device property files created for an additional device should be copied to the folder in the J2ME Wireless Toolkit's installation that contains device definitions. The new device is automatically added the device list.

Once the toolkit is installed, and A830.zip is downloaded, follow step 3 above. To add the Motorola A830 handset to the toolkit device list, unzip A830.zip to the {installed dir}\ wtklib\devices\A830\ directory. When the toolkit starts again, the Motorola A830 handset can be selected from the device pull-down menu in KToolbar.

If A830 does not show up in the device list, please make sure that:

1. The toolkit was restarted after unzipping A830.zip.
2. The name of the device directory matches the name of the device property file. For example, the name of the device should be A830 and the property file name should be A830.properties.
3. The property files and images should not be in any sub-directories under a device directory.

If the problem continues, please contact J2MEWTK-comments@sun.com.

Using Stubbed-out Classes

The Motorola A830 handset's developer support material package includes the A830 OEM classes for the developers to develop and test their application within the Wireless Toolkit. These classes behave similarly to or the same as in the Motorola A830 handset; however, some of the functionalities of some classes have been removed because they can not be simulated within the toolkit. A object, such as `com.motorola.location`.PositionSource, represents a connection interface on the Motorola A830 handset*; however, in the emulator, it only allows a MIDlet to compile and simulate its functionality using the `System.out.println()` method. Thus, in the emulator, when a MIDlet requests a connection interface within its application, the stubbed-out PositionSourcer object will output "The connection was obtained successfully." to the console. Similar functionalities have been adapted to the other A830 OEM classes.

These external stubbed-out libraries can be added to the `apps/{project.name}/lib/` or `apps/lib/` directories in zipped format. Refer to item 0 for detailed information on directories within the toolkit. For more information, please refer to *UserGuide.pdf* in `docs\` directory.

Packaging Applications

One of the downfall of placing external class libraries in the `apps/{project.name}/lib/` or `apps/lib/` directory is that when the MIDlet is packaged using the toolkit, it adds the project's class files as well as all external class libraries into the JAR file. Since the KVM is running on the Motorola A830 handset already, the MIDlet does not require to package stubbed out class files. Please use the following steps to remove external library classes from the JAR file before downloading the MIDlet onto the Motorola A830 handset:

1. Open the JAR file using WinZip or any other application that supports JAR.
2. Select and remove all external class libraries from the JAR file.
3. Open the JAD file and change the value of the MIDlet-Jar-Size attribute based on the new file size of the JAR file.
4. Save and close the JAD file.
5. Now the MIDlet is ready to be downloaded.

For additional information on packing MIDlets, please refer to *UserGuide.pdf* in the `docs\` directory.