

Preface

Dear Students,

This book introduces, shortly & effectively, the Java 2 Micro Edition (J2ME), the Java edition for small devices. The J2ME is divided into configurations and profiles. A configuration is designed for a specific kind of device based on memory constraints and processor power. A profile is more specific than configuration. The profile is based on a configuration and adds more APIs to it. This book concentrates upon the CLDC configuration (for very small devices) and the MIDP profile (designed for mobile information devices, such as mobile telephones).

Good Luck,

Saiful Adli Ismail.

Table of Contents

Chapter 1:	<u>Let's Start</u>
Chapter 2:	<u>The MIDlet</u>
Chapter 3:	<u>GUI</u>
Chapter 4:	<u>Persistent Storage</u>
Chapter 5:	<u>Networking</u>
Chapter 6:	<u>The Canvas class</u>
Chapter 7:	<u>Performance</u>
Chapter 8:	<u>XML</u>

Chapter 1:

Let's Start

- Java Benefits for wireless devices
- J2ME, J2SE and J2EE
- Configurations and Profiles
- The MID Profile
- The KVM
- Writing the first MIDlet
- J2ME (CLDC\MIDP) comparing to the J2SE

Java benefits for wireless devices

Java is ideally suited to become the standard application development language for wireless devices, providing us with lots of benefits. Hereto some of the most important ones:

Cross platform compatibility

The Java application can easily transferred between different devices and different platforms as long as the JVM\KVM has been developed for those devices.

Object Oriented Programming

Java has a better abstraction mechanisms and higher level programming constructs than C++.

Huge java developer community

Java has become the most popular programming language taught in schools and universities.

Security

Java is known for its security features (class file verification, cryptography possibilities etc...)

Dynamic

Java classes can easily downloaded dynamically over the network, and easily integrated with the running application.

J2ME, J2SE and J2EE

Sun Microsystems has grouped the Java Technologies into three editions that aimed at the specific areas of today's software industry:

Java 2 Platform, Enterprise Edition (J2EE) for enterprises needing scalable server solutions.

Java 2 Platform, Standard Edition (J2SE) for the personal computer (desktop).

Java 2 Platform, Micro Edition (J2ME) for small devices such as: PDA's, mobile telephones, pagers, consumer devices etc...

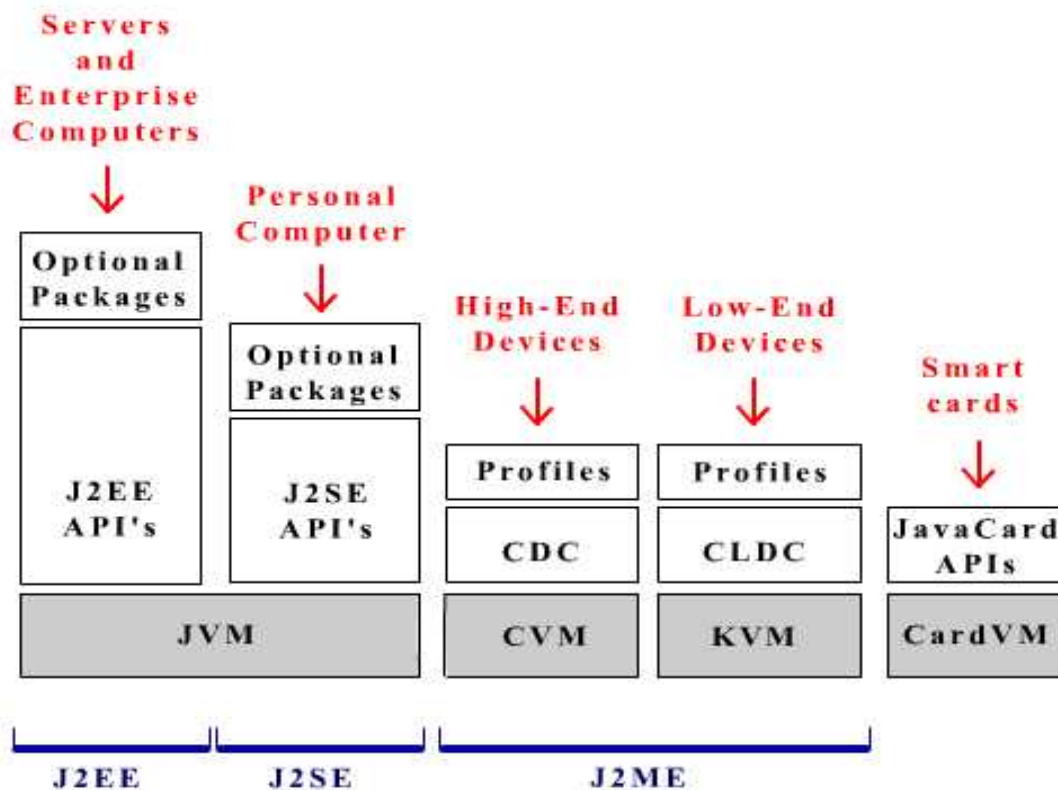
Each Java Platform edition defines a set of technologies:

JVM

Libraries of classes\API's

Tools for development

The following figure illustrates the Java 2 Platform editions and their target markets.



The J2ME has two configurations:

CDC (Connected Device Configuration)

and

CLDC (Connected, Limited Device Configuration)

These two configurations target two categories of products:

High-end devices (TV set-top boxes, Internet TVs, automobile navigation systems etc...) which have large range of user interface capabilities, high-bandwidth network connections (usually TCP/IP) and more memory and CPU power.

and

Low-end devices (Cell phones, pagers, PDA's etc...) which have very simple user interface and small screen size, less memory, less CPU power, lower bandwidth and most of them are usually operated using batteries.

Configurations and Profiles

Using the configurations and the profiles the J2ME platform can be expanded and customized as needed.

Configuration

A J2ME configuration defines the Java language, the virtual machine features and the minimum class libraries for a category\group of devices ("horizontal" market).

Profile

A J2ME profile is layered on top of configuration and address the specific demands of a specific "vertical" market (or device category). The profile typically includes class libraries that are far more specific than the class libraries provided in a configuration.

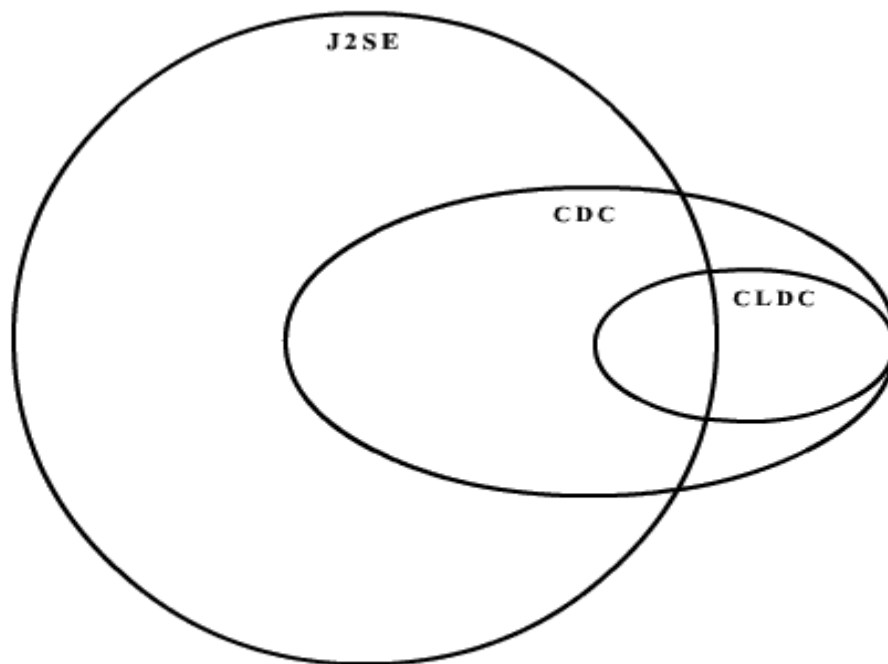
A device can support multiple configurations and therefore it can also support multiple profiles. Therefore, the vision of taking a specific application for mobile telephone and run it on the TV set top box (given that it supports the MID profile in addition to its "normal" more device specific profile) can become a reality.

A J2ME application is written for a specific profile. The profile is "based upon" a particular configuration and because of that all of the features of that

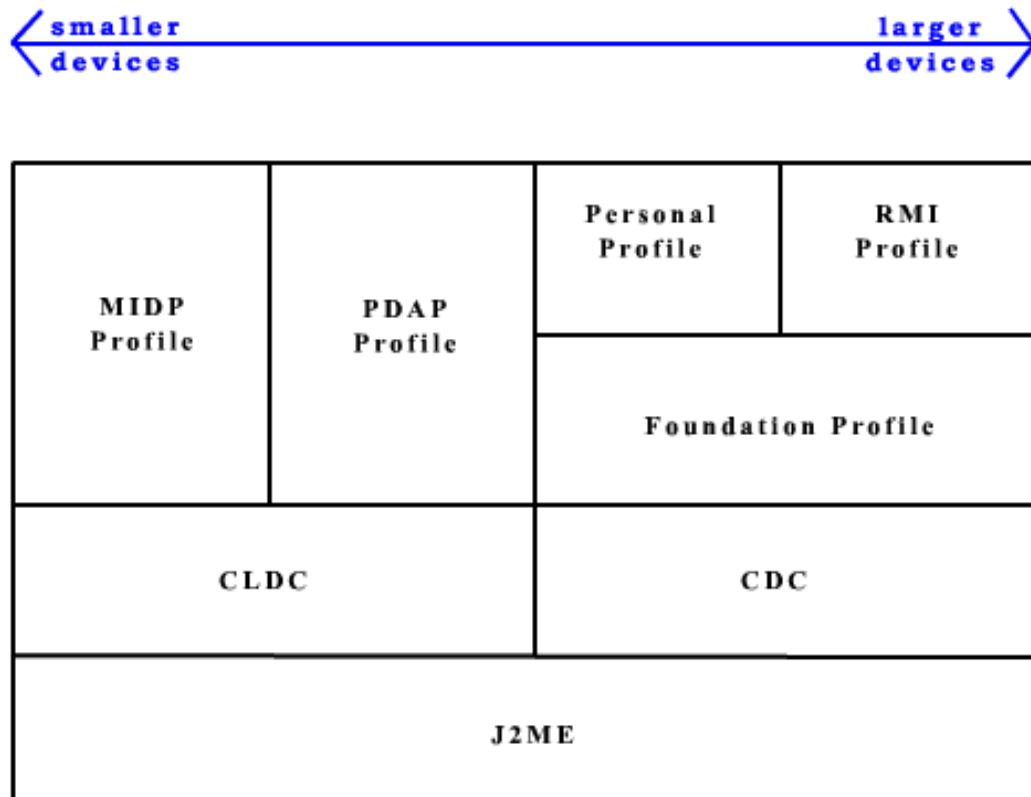
configuration are automatically included in the profile. Manufacturers choose which profile(s) to support on each of their devices and once they choose the profile(s) they must implement all the features of that/those profile(s).

Currently there are only two standard J2ME configurations:

Connected, Limited Device Configuration (**CLDC**) and the Connected Device Configuration (**CDC**). The classes that the CLDC and CDC configurations have might be inherited from the J2SE environment or a subset of these classes or new ones that don't exist in J2SE. The following figure demonstrates the relationships between the CDC, CLDC and the J2SE.



The MID profile is only one profile. More profiles are on their way to be published. The following figure demonstrates the profiles and the configurations relationships.



This book focus in the CLDC configuration and MIDP(the MID profile). The MID profile was designed for mobile information devices such as cellular telephones, PDA's and pagers.

The MID Profile

According to the specification, the MID profile has the following characteristics:

At least 128KB of non-volatile memory (non-volatile memory is memory that is capable of keeping its contents intact as the device is turned on and off. The ROM is one example for non-volatile memory) for the MIDP implementation.

At least 32KB of volatile memory for the heap.

At least 8KB of non-volatile memory for persistent data.

A screen of at least 96x54 pixels.

Pixel ratio approximately 1:1.

An Input mechanism - one of the following: keypad, keyboard or touch screen.

Two-way network connection.

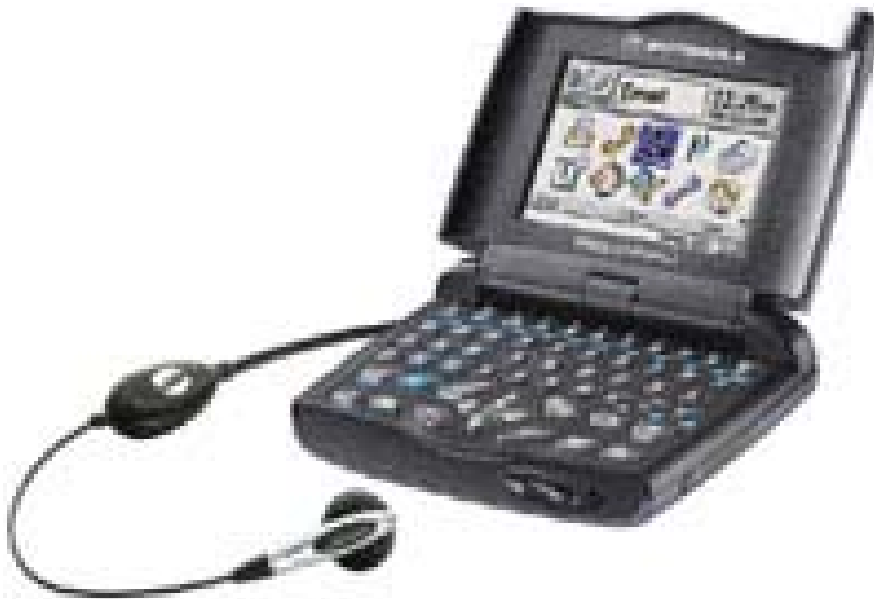
The classes a MIDP application can use come from packages in both the CLDC and the MIDP. The packages the CLDC has are: `java.lang`, `java.io`, `java.util`, `javax.microedition.io`. The packages the MID profile adds are: `javax.microedition.lcdui`, `javax.microedition.midlet` and `javax.microedition.rms`.

Once a MID software is developed it can be categorized to one of the following three types of MID software:

1. **Native Application** which was compiled from C/C++ or other native language and runs directly on the device operating system. This kind of application is developed specifically for a specific device and it can run only on that device.

2. **MIDP java application** that was written in Java and that uses only the MIDP & CLDC APIs. This kind of application can run on any device that supports the MID profile.
3. **Device-Specific application** that was written in Java and uses in addition to the MIDP & CLDC APIs some specific API(s) that were\was developed specifically by the device vendor. These type of applications won't run on other devices that don't have those extra APIs.

The following pictures present several mobile telephones that support the J2ME (CLDC\MIDP). For more information about mobile telephones that support the J2ME technology visit <http://www.javamobiles.com> .



















The KVM

The CLDC and the MID profile run on top of Sun's K Virtual Machine (KVM), which is a compact, portable JVM specifically designed for small, resource-constrained devices. The "K" in KVM stands for "kilo". It was so named because its memory is measured in tens of kilobytes. The currently minimum memory budget required by a KVM implementation is about 128 K (including the Virtual Machine, minimal libraries and some heap space).

In some devices the KVM is used on top of existing native software to give the device the ability to run Java content.

In other devices the KVM is used at lower level, implements the lower-level system and serves as the device main software that responsible for the device main activities.

Writing the first MIDlet

Writing a MIDlet is similar to writing an applet. The MIDlet we write is a class that extends the `Midlet` class (similar to declaring a class that extends `Applet`). To develop a MIDlet you will have to download the "Java 2 Micro Edition Wireless Toolkit" from www.javasoft.com. The code in this book were tested and run using the "Java 2 Micro Edition Wireless Toolkit 1.0.2 Early Access 2". You can also find other alternative toolkits (The Motorola SDK, for instance). After all, the MIDP is only a specification that different vendors can adopt and develop their own implementation. The "Java 2 Micro Edition Wireless Toolkit" (J2MEWTK) contains a reference implementation of the MID profile as well as some other software tools that were designed especially for developing MIDP applications.

Coding:

Writing the source code of the MIDlet is the same like any other java code. Code the following simple example using your favorite text editor:

```
//filename:HelloMalaysia.java
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;
```

```

public class HelloIsrael extends MIDlet
{
    private Display display;
    private Form form;

    public HelloIsrael()
    {
        form = new Form("Hello Malaysia");
        display = Display.getDisplay(this);
    }

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

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
    }
}

```

Compiling:

When compiling the source code you should use the classes that belong to the MID profile implementation instead of the classes that belong to the J2SE (that you should have installed on your computer). To do so, you should point to the path in which the MIDP classes reside. You should write the following (in the command line):

```
javac -bootclasspath classesPath HelloMalaysia.java
```

The `-bootclasspath` is one of the options that can be sent to the `javac` command when invoking it. This option is used to point out at the location of the bootstrap class files. Given that you use the J2MEWTK, and given that you have installed it with its default folder names in disk C, you should write the following:

`javac -bootclasspath c:\j2mewtk\lib\midpapi.zip HelloMalaysia.java`

Alternatively, you can start the KToolbar utility application (part of the J2MEWTK). Under the "bin" folder you shall find the "KToolbar.bat" file that starts the KToolbar application. When it starts, choose the "New Project" option from its toolbar and create a new project (name it as you want). As a result of this step the "src" folder is created within a folder named after the project name you chose. The "HelloMalaysia.java" file is saved in the "src" folder. Now, you can just choose the option "build" from the menu bar and let the KToolbar does the compilation job for you.

Preverifying:

Unlike the J2SE, when working with the CLDC configuration, the bytecode verification process is divided into two processes. The first takes place off the device (this is the preverifying step you should take) and the second process takes place on the device itself (the device just needs to do lightweight second verification before loading the classes). If the device gets a class that has not been preverified it is rejected.

The J2MEWTK contains a tool called "preverify" that performs the first step (the preverify.exe file resides in the "build\win32\tools" directory. It is recommended adding this directory to the path environment variable). The result of the preverify utility program is a new *.class file. Given that your current directory is the one in which the HelloIsrael.class was saved, all you have to type in the command line is:

`preverify -classpath .:c:\j2mewtk\lib\midpapi.zip -d . HelloMalaysia`

In this case, the -d option tells preverify to write the preverified class file to the current directory. If you chose to use the KToolbar tool, then the preverifying step is done automatically for you, when you press the "build" button.

Running:

If you chose to use the KToolbar tool, then you can simple press the "run" button.

If you chose to work from the command-line, and given that the emulator.exe file resides in c:\j2mewtk\bin directory and the current directory is the one in which the HelloMalaysia class resides, then you should do the following:

```
c:\J2mewtk\bin\emulator      -classpath      .;c:\j2mewtk\lib\midpapi.zip  
HelloMalaysia
```

An alternative way for testing your MIDlet is by creating a .jad file that describes the MIDlet application and invoke it using the emulator utility application. The alternative way won't be described at the moment.

The MIDlet life cycle is controlled by the **JAM** (Java Application Manager), a MIDlet management software that controls the process of installing, running and removing the MIDlets. When the user chooses to run a MIDlet, it is the JAM that creates an instance of the MIDlet class and runs methods on it. The sequence of method that will be invoked on the MIDlet subclass instance is defined by the MIDlet life cycle. Like the servlets and the applets, the midlets also have a well-defined set of states.

The JAM calls the different methods on the midlet to signify changes from one state to another. These methods include the following: startApp(), pauseApp(), destroyApp() and the MIDlet subclass constructors as well.

J2ME(CLDC\MIDP) comparing to the J2SE

The MID profile is built on top of the CLDC configuration. Some of the CLDC APIs is based on the basic APIs of the J2SE. The differences between the two (the CLDC and the J2SE) are as follows:

The J2ME(CLDC\MIDP) doesn't support floating-point types.

The J2ME(CLDC\MIDP) doesn't have the float and double primitive types and not the Float and Double classes.

The J2ME(CLDC\MIDP) java.lang package is a subset of the J2SE java.lang package.

The classes and interface that the J2ME(CLDC\MIDP) java.lang package has are: Runnable, Boolean, Byte, Character, Class, Integer, Long, Math, Object, Runtime, Short, String, StringBuffer, System, Thread, Throwable, ArithmeticException, ArrayIndexOutOfBoundsException, ArrayStoreException, ClassCastException, ClassNotFoundException, Exception, IllegalAccessException, IllegalArgumentException, IllegalMonitorStateException, InterruptedException, IndexOutOfBoundsException, InstantiationException, InterruptedException, NegativeArraySizeException, NullPointerException, NumberFormatException, RuntimeException, SecurityException, StringIndexOutOfBoundsException, Error, OutOfMemoryError and the VirtualMemoryError. Some of these classes don't have the same API the J2SE classes have (they are subset to the J2SE classes).

The J2ME(CLDC\MIDP) doesn't enable controlling the Class Loading process.

The J2ME(CLDC\MIDP) doesn't allow you defining your own class loader. The JAM has a class loader that doesn't allow accessing\controlling it.

The J2ME(CLDC\MIDP) doesn't support Object finalization mechanism.

As you probably remember from the J2SE platform, using the Finalization mechanism has never been recommended. Therefore, not having the support of this mechanism can only be count as an improvement.

The J2ME(CLDC\MIDP) doesn't support the reflection mechanism.

Not having the support of the reflection mechanism in the J2ME(CLDC\MIDP) profile has severe implications. Because of that we can't use RMI and therefore we can't use JINI :(. However, using one of the other J2ME profiles (such as the RMI profile) can be a solution.

The J2ME(CLDC\MIDP) doesn't support a mechanism (like the JNI) that enables writing native methods.

If you decide to use the MID profile there is no way of combining the java code with other languages code.

The J2ME(CLDC\MIDP) multithreading mechanism is very similar to the one that the J2SE has.

As long as the multithreading isn't complicated you will find the J2ME(CLDC\MIDP) multithreading support sufficient. Most of the deprecated methods from the J2SE edition, resume(), suspend() and stop() are not supported. The thread grouping mechanism and the daemon threads also are not supported.

The J2ME(CLDC\MIDP) has the Math class

The J2ME(CLDC\MIDP) Math class API is a subset of the J2SE version. Most of the methods, that the J2SE Math class has, don't exist in the J2ME version.

The J2ME(CLDC\MIDP) has smaller version of the String and the StringBuffer classes

The J2ME(CLDC\MIDP) String and StringBuffer classes are very similar to the J2SE versions. Only few methods were omitted in the transition from J2SE to J2ME(CLDC\MIDP).

The J2ME(CLDC\MIDP) has smaller version of the System and the Runtime classes

Lots of methods that exist in the J2SE don't exist in the J2ME(CLDC\MIDP). One example is the exec() method that belongs to the Runtime class (in its J2SE version) and doesn't exist in the J2ME(CLDC\MIDP) version since the MIDP application can't run out of the JVM's bounds.

The J2ME(CLDC\MIDP) java.util package include only few of the classes and interface that exist in the J2SE version.

The classes and the interfaces that the J2ME(CLDC\MIDP) supports in its version of the java.util package are: Enumeration, Calendar, EmptyStackException, NoSuchElementException, Date, Hashtable, Vector, TimeZone, TimerTask, Timer, Stack and Random. One of the biggest changes is the missing of most of the collection classes.

The new Timer and TimerTask classes that were introduced in the J2SE 1.3 version were included in the J2ME (CLDC\MIDP).

The J2ME(CLDC\MIDP) java.io package include only few of the classes and interfaces that exist in the J2SE version

The classes and the interface that the J2ME (CLDC\MIDP) supports in its version for the java.io package are: InputStream, OutputStream, ByteArrayInputStream, ByteArrayOutputStream, DataInputStream, DataOutputStream, Reader, Writer, InputStreamReader, OutputStreamWriter, PrintStream, EOFException, InterruptedIOException, IOException, UnsupportedEncodingException, UTFDataFormatException and the DataOutput and DataInput interfaces.

The biggest changes are the serialization mechanism that the J2ME(CLDC\MIDP) doesn't support and the files streams (The local file system doesn't exist in MIDP devices).

The encoding mechanism still exists. However, in the J2ME(CLDC\MIDP) there are fewer available encoding than in J2SE. You can get the default encoding by calling the `System.getProperty("microedition.encoding")` method.

The J2ME (CLDC\MIDP) still supports the resource files idea (as long as these files are packaged within the MIDlet suit's JAR file). To get an input stream connected to specific resource file you can use the `getResourceAsStream()` method that was declared in the `Class` class.

Chapter 2:

The MIDlet

- • [The MIDlet Life Cycle](#)
- • [MIDlet Suites](#)
- • [MIDlet Properties](#)
- • [The MIDlet suite Environment](#)
- • [The getResourceAsStream method](#)
- • [The JAM](#)

The MIDlet Life Cycle

When we develop a MIDlet, we simply develop a class that extends the MIDlet class and let the JAM (Java Application Manager) instantiate it and invoking different methods on it. The different methods that the JAM invokes on the midlet change the midlet state. As a result of that, the midlet goes through different states in its life cycle. The midlet states during its life cycle are as follows:

When the JAM receives the class (a class we declared as one that extends the MIDlet class) it instantiates it and positions it in the **Paused** state.

When the JAM invokes the startApp() method (on the midlet) the midlet moves to the **Active** state.

While the midlet in its **Active** state the JAM can invoke the pauseApp() method on it and by doing so, moves it back to the **Paused** state. If the active midlet wants to return back to the **Paused** state it can call the notifyPaused() method. By doing so, it will be moved back to the **Pause** state.

While the midlet in its **Active** state the JAM can invoke the destroyApp() method on it and by doing so, moves it to the **Destroyed** state (in other words: destroy it). If the active midlet wants to be **Destroyed** it can do it self by calling the notifyDestroyed() method. Calling the notifyDestroyed() method doesn't invoke the destroyApp() method. The midlet must have performed the same operations (clean up, releasing of resources etc...) it would have if the MIDlet.destroyApp() had been called. Once a midlet enters the Destroyed state, it cannot reenter any other state.

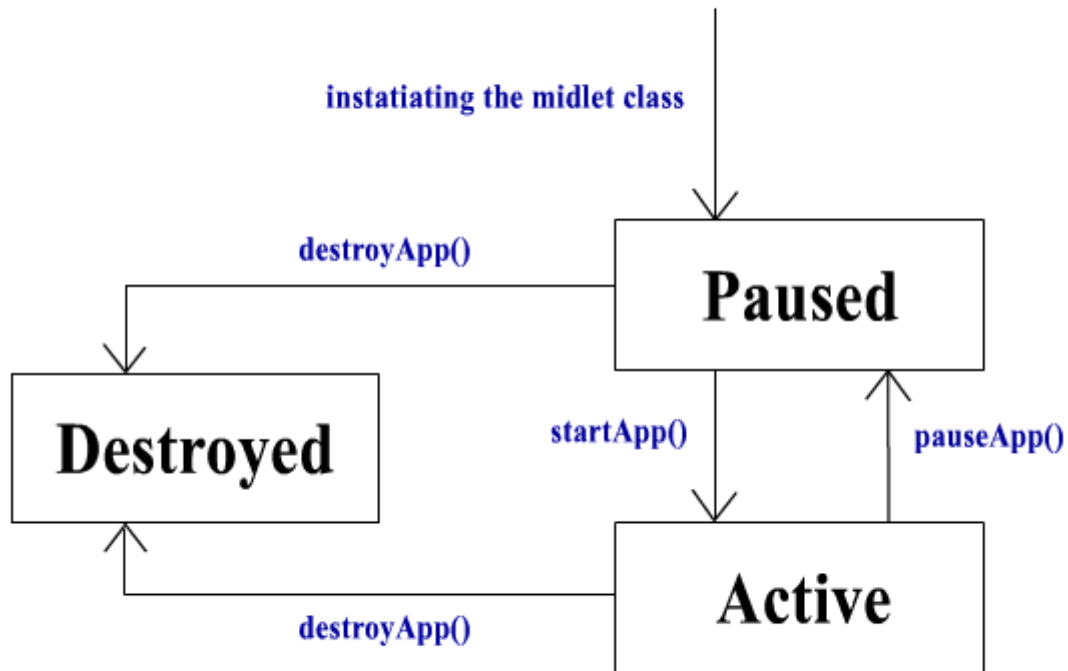
The midlet in its Paused state can call the `resumeRequest()` method and signal (by doing so) to the JAM that it wants to become active again. In response to that the JAM will call the `startApp()` method on that midlet.

The `startApp()` method might get called more than once. Therefore, any operations that must be performed only once when the application is launched should be placed in the constructor. The `startApp()` method may be called by the system under different circumstances and its purpose is to prepare the midlet to handle different events as well as reacquire the needed resources. Each time, the midlet is resumed the `startApp()` method is called.

The `pauseApp()` method job is usually releasing as many resources as possible toward the next midlet restart (when the `startApp()` method is invoked). Placing nulls in variables the midlet has will cause indirectly the garbage collector to free these memory spaces. When the midlet is in the Paused state it should hold as few resources as possible. A midlet in the Paused state can receive asynchronous notifications (from a timer firing, for instance).

The `destroyApp()` method job is performing all the necessary clean up operations to release all the resources the midlet had allocated during its execution (closing network connections, database records etc...). The `destroyApp()` method has one boolean parameter that indicates whether the midlet termination is unconditional (true) or not. If the boolean parameter's value is false then the midlet might throw `MIDletStateChangeException` and by doing so, indicate the device that it wants to stay alive.

The following diagram demonstrates the different states in which the midlet can be.



The following midlet demonstrates the different states in which the midlet can be. Even though the `System.out.println()` commands are not seen when the midlet runs on a real device, when the midlet is tested using one of the existing emulators the `System.out.println()` commands are seen in the command line. Parts of the following example deal with GUI and events handling. These topics will be explained later.

```
//filename:HelloMalaysia.java
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;
```

```

public class MIDletStates extends MIDlet implements CommandListener
{
    private Display display;
    private Command exitCommand;
    private TextBox textBox;

    public MIDletStates()
    {
        System.out.println("The MIDletStates constructor is invoked");
        exitCommand = new Command("Exit", Command.EXIT, 1);
        textBox = new TextBox("MIDlet's states demo", "Hello:", 15,0);
        textBox.addCommand(exitCommand);
        textBox.setCommandListener(this);
    }

    public void startApp()
    {
        System.out.println("The startApp method is invoked");
        display = Display.getDisplay(this);
        display.setCurrent(textBox);
    }

    public void pauseApp()
    {
        System.out.println("The pauseApp method is invoked");
    }

    public void destroyApp(boolean cond)
    {
        System.out.println("The destroyApp method is invoked");
    }

    public void commandAction(Command command, Displayable displayable)
    {
        if(exitCommand==command)
        {
            destroyApp(true);
        }
    }
}

```

MIDlet Suites

A MIDlet suite is composed of two files.

The first is a simple text file (.jad) that describes the MIDlet suite.

The second is a JAR file that contains all the class files as well as the resource files needed for the midlet and a MANIFEST file.

The steps in getting a MIDlet suite are the following. First, the class files as well as the resource files are packaged into one JAR file. Together with these files, a MANIFEST file is packaged as well. After the JAR file was created, the MIDlet suite descriptor file (*.jad) is created. The JAM uses the jad file in its decisions.

The MIDlet manifest file consists of name and value pairs and should include the following attributes:

MIDlet-Name - The name of the entire MIDlet suite.

MIDlet-Version - This describes the version of the MIDlet suite. You choose this number. The format of this attribute is as described in the JDK Product Versioning Specification. The format is major.mino.micro (1.1.2 for instance). The value of the MIDlet-Version attribute ca be used by the system for installation and upgrade uses.

MIDlet-Vendor - This is the name of the company that developed this MIDlet suite.

MIDlet-n - The MIDlet suite can holds more than one MIDlet. Each MIDlet in the suite can have a number(starting at 1), the displayable name(this name will be used by the user), icon file (PNG file) and the class name. The MIDlet in the MIDlet suite should be numbered starting from 1 and counting up.

MicroEdition-Configuration - The J2ME configuration needed to run the MIDlet suite. CLDC-1.0 will be the MicroEdition-Configuration when using the MIDP1.0.

MicroEdition-Profile - The profile required by this MIDlet suite. When using the MIDP1.0, the value of this attribute should be MIDP1.0.

More attributes can be added in addition to the required manifest attributes:

MIDlet-Data-Size - The number of bytes of persistent data that are required by the MIDlet suite.

MIDlet-Description - Textual description of the MIDlet suite.

MIDlet-Icon - The icon to represent the entire MIDlet suite. The file is a PNG one.

MIDlet-Info-URL - The URL that has additional information about the MIDlet suite.

Manifest-Version - The manifest file version.

Given that the HelloIsrael class from chapter 1 belongs to the com.zindell.j2me package, the following can be the manifest file of that HelloIsrael MIDlet:

```
Manifest-Version: 1.0
MIDlet-1: HelloIsrael, HelloIsrael.png, com.zindell.j2me.HelloIsrael
MIDlet-Name: HelloIsrael
MIDlet-Version: 1.0
MIDlet-Vendor: ZINDELL
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-1.0
```

The JAM can manage more than one MIDlet. The information within the JAR file (especially in its MANIFEST files) helps him in doing so.

The MIDlet suite descriptor file is a file that has a MIME type of text/vnd.sun.j2me.app-descriptor and a file extension of **.jad**. The device uses the jad file in its decision whether loading the MIDlet suite or not.

Some of the information that the application descriptor file (the jad file) contains exists in the MANIFEST file as well. The attributes the jad file must have are:

MIDlet-Name

MIDlet-Version

MIDlet-Vendor

MIDlet-Jar-URL

MIDlet-Jar-Size

The jad file can also have the other attributes the MANIFEST file can have.

Given that the HelloMalaysia class from chapter 1 belongs to the com.zindell.j2me package, the following can be the jad file of that HelloMalaysia MIDlet suite:

Manifest-Version: 1.0

MIDlet-1: HelloMalaysiaProject, HelloMalaysiaProject.png,
com.zindell.j2me.HelloMalaysia

MIDlet-Jar-Size: 1056

MIDlet-Jar-URL: HelloMalaysiaProject.jar

MIDlet-Name: HelloMalaysiaProject

MIDlet-Vendor: ZINDELL

MIDlet-Version: 1.0

If the MIDlet-Name, MIDlet-Version and the MIDlet-Vendor attributes are not identical in the MANIFEST and the jad files then the MIDlet suite won't be installed. If the other attributes that exist in the two files (The MANIFEST & the jad files) then the attributes in the jad file will count.

MIDlet Properties

It is possible adding attributes and their values as well to the manifest file or\and the application descriptor file. Later, the midlet will be able retrieving these attributes' values using the `getAddProperty()` method, that was declared in the MIDlet class. Placing these attributes in the application descriptor file (the jad file) enables a better separation between the coding and the deployment. The attribute names are case sensitive.

The following example presents the use of this mechanism. The following midlet needs to have in its jad file the property: `MIDPSuiteProperties.toWhom` with a value the midlet will retrieve and display. Working with the KToolbar easiest the creation of the MIDlet suite descriptor file (the jad file).

```
//filename:HelloMalaysia.java
```

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
```

```
public class HelloToWhom extends MIDlet
{
    private Display display;
    private Form form;

    public HelloToWhom()
    {
        String str = getAppProperty("MIDPSuiteProperties.toWhom");
        form = new Form(str);
    }

    public void startApp()
    {
```

```

        display = Display.getDisplay(this);
        display.setCurrent(form);
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
    }
}

```

The MIDlet suite Environment

The MID profile specification defines the MIDlet suite environment.

Only the classes and the native code that implement the CLDC and MIDP are shared by all of the MIDlet suites on the device.

The MIDlet suite is the basic unit of J2ME(CLDC\MIDP) application. The class files (the different midlets) as well as the resource files within the MIDlet suite can't be manipulated (installed, updated or removed) individually. They must be manipulated as a whole. The MIDP specification does not allow for individual MIDlets to be upgraded to newer version. The entire MIDlet suite must be upgraded as a unit. Thanks to that, the original intent of the MIDlet suite provider is not changed.

The MIDlet suite has a common name space for its midlets. This common name space is used for the Object Heap and the static fields (of the different classes)

only. Each midlet can interact with the other midlets that reside in the same MIDlet suite.

The record stores have separated name spaces (The RMS topic will be introduced later).

The class files can't be manipulated (can't be read nor extracted for re-use). The other non-class files within the JAR file can be accessed using the `java.lang.Class.getResourceAsStream` method:

```
public InputStream getResourceAsStream(String name)
```

The `getResourceAsStream` method

The other non-class files within the JAR files can be accessed using the `java.lang.Class.getResourceAsStream` method:

```
public InputStream getResourceAsStream(String name)
```

This method finds a resource with a given name. This method returns null if no resource with this name is found. The rules for searching resources are as follows: If name starts with a '/', the search for the resource begins at the 'root' of the JAR file. If it doesn't begin with a '/', the resource is searched for along a path relative to the class instance retrieving the resource.

Given the `details.txt` file consists of the following:

ZINDELL.COM

The following example demonstrates the use of the `getResourceAsStream()` method.

```
//filename:ResourceDemo.java
```

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
```

```
public class ResourceDemo extends MIDlet
{
    private Display display;
    private Command exitCommand, stopCommand;
    private TextBox textBox;

    public ResourceDemo()
    {
        try
        {
            Class theClass = Class.forName("ResourceDemo");
            InputStream is = theClass.getResourceAsStream("/details.txt");
            StringBuffer sb = new StringBuffer();
            int temp = is.read();
            while(temp!=-1)
            {
                sb.append((char)temp);
                temp = is.read();
            }
            String str = sb.toString();
            System.out.println("str="+str);
            textBox = new TextBox("Welcome To", str, 15,0);
        }
        catch(Exception e)
        {
        }
    }

    public void startApp()
    {
        display = Display.getDisplay(this);
        display.setCurrent(textBox);
    }
}
```

```
    }  
  
    public void pauseApp()  
    {  
    }  
  
    public void destroyApp(boolean cond)  
    {  
    }  
}
```


The JAM

The JAM (Java Application Manager) or in its other name, the Application Management Software, in the MIDP specification consists of different software pieces on the device that provide a framework in which the MIDlet suites are managed. The typical application management operations include the following:

Retrieval

Retrieving a MIDlet suite. This can be done from PC, wireless network, infrared connection or another source according to what the device enables.

Installation

Installing the MIDlet suite on the device. This step might include some kind of verification process.

Launching

Launching the midlet on the device (instantiating the midlet class).

Version Management

Installed MIDlet suites have versions. The JAM manage its MIDlet suites and enable their upgrade to newer versions.

Removal

Removing MIDlet suites that were installed on the device. The deletion of the MIDlet suite includes the deletion of its resource files.

Chapter 3:

GUI

- • [Introduction](#)
- • [The Display and the Displayable classes](#)
- • [Images](#)
- • [Events Handling](#)
- • [List](#)
- • [TextBox](#)
- • [Alert](#)
- • [Form and Items](#)
- • [ItemStateChanged Events](#)

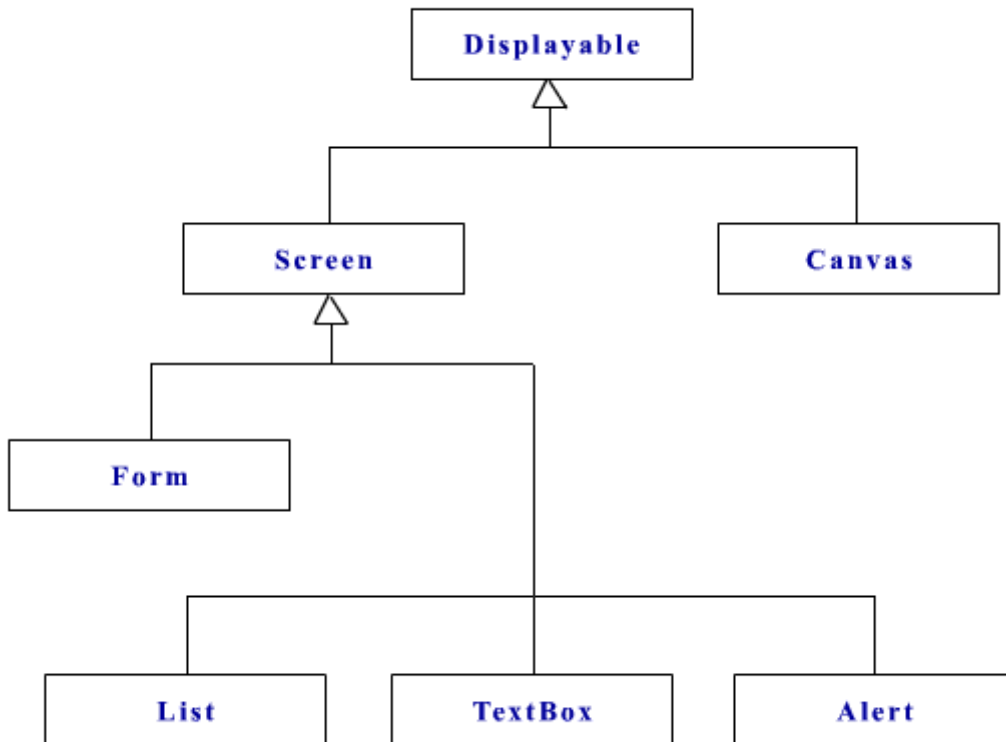
Introduction

Creating a standard GUI for all the platforms that support the use of J2ME is a big problem. Each device might have a different screen size and different resources. Instead of having the same GUI mechanism for every device that support the J2ME, each profile has a different GUI mechanism, one that is suitable for the devices it was planned for. Therefore, the GUI API for mobile information devices (cellular telephones for instance) is defined in the MID profile and in the same time, the GUI API for the PDAs (the Palm Pilot for instance) is defined in the PDAP profile.

The GUI classes for the MID profile are defined in the `javax.microedition.lcdui` package. The MID profile is characterized by its high level of abstraction (the actual drawing and user interaction are performed by the MIDP implementation). The J2ME(CLDC\MIDP) also enables some low level API actions that might be device-dependant and therefore should be avoided.

The Display and the Displayable classes

The **Displayable** class describes an object that encapsulates device-specific graphics rendering and therefore can be displayed to the user. The Displayable class is the base class for many other classes. The following figure presents it.



This tree of classes presents three types of Displayable objects:

The **Canvas** class enables full control of the appearance and accessing low level of events.

The **List**, **TextBox** and the **Alert** classes describe predefined structured components. Each instance of these classes is also a **Screen**, which means that it takes over all the screen area. It isn't possible adding other displayable objects to object of one of these classes.

The **Form** class describes a component(screen) that can be populated with text, images and other simple sets of component.

The Displayable class has three methods:

```
public void addCommand(Command cmd)
```

```
public boolean isShown()
```

```
public void removeCommand(Command cmd)
```

```
public void setCommandListener(CommandListener listener)
```

The *Command* and *CommandListener* classes will be discussed later, when the events handling will be explained.

The *Screen* abstract class is the super class of the *Alert*, *Form*, *List* and *TextBox* classes. A *Screen* instance can contain an optional title as well as a ticker-tape (a text that continuously runs across the screen).

The *Canvas* abstract class extends the *Displayable* class and implements the low-level API. In order to use this class there is a need in sub classing it. More details regarding the use of the *Canvas* class will be brought later.

The Display class represents the device display manager. Using an object of this class it is possible retrieving properties of the device and for setting a specific object as the displayed one.

The *Display* class is singleton. The methods the *Display* class has are:

```
public static Display getDisplay(MIDlet m)
```

This method returns the one and only *Display* object that *m MIDlet* has.

```
public boolean isColor()
```

This method returns true if the device supports colors.

```
public int numColors()
```

This method returns the number of colors the device supports.

```
public Displayable getCurrent()
```

This method returns the current Displayable object. Each moment, only one Displayable object can be displayed. The displayable object might be invisible if the midlet runs in the background or if the device decides on showing some other specific system screen.

```
public void setCurrent(Displayable nextDisplayable)
```

This method set the current Displayable object. Each moment, only one Displayable object can be displayed. The displayable object might be invisible if the midlet runs in the background or if the device decides on showing some other specific system screen. The change typically does not take effect immediately. The change might be delayed so that it occurs between event delivery methods calls.

```
public void setCurrent(Alert alert, Displayable nextDisplayable)
```

Requests that this Alert will be the current Displayable, and that nextDisplayable be made current after the Alert is dismissed. The nextDisplayable must not be an Alert, and it must not be null.

In other respects, this method behaves identically to setCurrent(Displayable).

```
public void callSerially(Runnable r)
```

Causes the Runnable object `r` to have its `run()` method called later, serialized with the event stream, soon after completion of the repaint cycle.

Each moment, only one Displayable object can be displayed. The Display object is the one that responsible for displaying a Displayable object (a form, a canvas etc...).

The following example presents a simple advertisement for ZINDELL. The following midlet presents several forms that witched with each other every 1500 milliseconds. Each form displays a different text. This example also presents the use of the Ticker class. The midlet has the same Ticker object connected with each of the forms.

```
//filename:ZindellAd.java
```

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
```

```
public class ZindellAd extends MIDlet implements Runnable
{
```

```
    private Display display;
    private Form forms[];
    private Ticker ticker;
    private int counter = -1;
```

```
    public ZindellAd()
    {
```

```
        forms = new Form[4];
        forms[0] = new Form(" T R A I N I N G");
        forms[1] = new Form(" C O N S U L T I N G");
        forms[2] = new Form(" M E N T O R I N G");
        forms[3] = new Form(" D E V E L O P I N G");
```

```

        ticker = new Ticker("ZINDELL.COM gives the following services. call
+972+58+355837");
        forms[0].setTicker(ticker);
        forms[1].setTicker(ticker);
        forms[2].setTicker(ticker);
        forms[3].setTicker(ticker);
    }

    public void run()
    {
        while(true)
        {
            try
            {
                Thread.sleep(1500);
                if(counter==3)
                {
                    counter=-1;
                }
                counter++;
            }
            catch(Exception exception)
            {
                exception.printStackTrace();
            }
            display.setCurrent(forms[counter]);
        }
    }

    public void startApp()
    {
        display = Display.getDisplay(this);
        Thread thread = new Thread(this);
        thread.start();
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        for(int i=0; i<forms.length; i++)
        {
            forms[i] = null;
        }
    }

```

```
        display = null;
        forms = null;
    }
}
```

Images

An Image object is used to hold graphical image data. The Image object can be painted to a Canvas or to be placed within a Form\Alert\List or ChoiceGroup element. The Image class enables the creation of two different types of images: immutable images and mutable images.

Immutable images cannot be modified once created. Images to be placed within Alert, Choice or ImageItem must be immutable. An immutable image can be created based on data that is stored in a resource file using the createImage() method.

```
.
.
Image img = Image.createImage("imageData.dat");
.
.
```

The data in the img file must be one of the supported formats. The PNG format must be supported. For now, the MIDP specification doesn't enforce the support of other format than PNG. Some devices support other formats as well and other don't.

Mutable images are like an offscreen canvas in which the midlet can paint once getting the Graphic object that represents them.

```
.
.
Image img = Image.createImage(200,100);
Graphics g = img.getGraphics();
```

.
.

The Image class has the following methods:

public static Image createImage(byte []vec, int imgOffset, int imgLength)

Creates an immutable image according to the data in vec.

public static Image createImage(Image src)

Creates an immutable image based upon src.

public static Image createImage(int width, int height)

Creates a new mutable image.

public static Image createImage(String name)

Creates an immutable image according to the data within the resource that name is its name.

public Graphics getGraphics()

Creates a new Graphic object that describes this image.

public int getHeight()

Returns the image height in pixels.

public int getWidth()

Returns the image width in pixels.

public boolean isMutable()

Returns true if this image is mutable.

The following midlet presents a simple use of the Image class.

```
//filename:PacMan.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;
```

```
public class PacMan extends MIDlet  
{  
    private Display display;  
    private Form form;  
  
    public PacMan()  
    {  
        form = new Form("P A C M A N");  
    }  
  
    public void startApp()  
    {  
        display = Display.getDisplay(this);  
        display.setCurrent(form);  
        try  
        {  
            form.append(Image.createImage("/PacManImg.png"));  
        }  
        catch(IOException exception)  
        {  
            form.append("problem in loading the picture");  
        }  
    }  
}
```

```

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
    }
}

```

Events Handling

The GUI defined in the MID profile is event driven. Events are generated in response to user interactions, and event handlers then process these events. The J2ME(CLDC\MIDP) uses the delegation model. Every object that is a Displayable can be an event source. The event listener is registered to the event source in order to listen to the events.

The MIDP GUI uses two types of events: The **Command** event and the **ItemStateChanged** event. Correspondingly, the MIDP GUI has two types of event listeners: The **CommandListener** and the **ItemStateListener**.

Every object that is a Displayable (was instantiated from a class that extends Displayable) can be the event source of the Command events.

Only Form instances can be the event sources of ItemStateChanged events.

The Command object encapsulates the semantic information of an action. The behavior that the command activates is not encapsulated in it, which means that the command object doesn't contain information about the actual action that happens when the command was activated. Each device might show the Command objects in a different way, and the way the Command objects are presented might also depend on the semantic information contained within the command.

The command contains three pieces of information:

Label

The string which is the visual representation of the command type

An integer that specifies the intent of this command - one of the predefined finals: `Command.BACK`, `Command.CANCEL`, `Command.HELP`, `Command.ITEM`, `Command.OK` etc...). The midlet uses the command type to specify the intent of this command. For example, if the command type is `BACK` and the device has a standard of placing the `BACK` operation on a certain soft-button, the implementation can follow that style. The command mapping is implementation dependent.

Priority

An integer that indicates the importance of the command. The bigger the number the less important the command. Typically, the implementation first chooses the placement of a command based on the type of command and then places similar commands based on a priority order. The application is always responsible for providing the means for the user to progress through different screens.

The constructor the Command class has is:

```
public Command(String label, int commandType, int priority)
```

The methods the Command class has are:

```
public int getCommandType()
```

This method returns the command type, which can be one of the predefined finals in this class.

```
public int getCommandType()
public String getLabel()
public int getPriority()
```

The predefined finals in the command class are:

```
public static final int BACK
public static final int CANCEL
public static final int EXIT
public static final int HELP
public static final int ITEH
public static final int OK
public static final int SCREEN
public static final int STOP
```

The following midlet presents a simple use of the Command class. This example just shows one command without handling the events.

```
//filename:ZindellAd.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class SimpleCommand extends MIDlet
{
```

```

private Display display;
private Form form;
private Command exitCommand;

public SimpleCommand()
{
    display = Display.getDisplay(this);
    form = new Form("simple exit command");
    exitCommand = new Command("EXIT", Command.EXIT, 1);
    form.addCommand(exitCommand);
}

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

public void pauseApp()
{
}

public void destroyApp(boolean condition)
{
    display = null;
    form = null;
}
}

```

The following midlet presents a simple use of the *Command* class. This example just shows several commands without handling the events. Try to run it with different emulators and see the different way each one of them presents the command. Remember that the command mapping is implementation dependent.

//filename:SimpleCommand.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
 //All rights reserved. No part of the contents of this program may be
 //reproduced or transmitted in any form or by any means without the
 //written permission of the publisher.

```

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class SimpleCommand extends MIDlet
{
    private Display display;
    private Form form;
    private Command exitCommand, yahooCommand, zindellCommand;

    public SimpleCommand()
    {
        display = Display.getDisplay(this);
        form = new Form("simple commands");
        exitCommand = new Command("EXIT", Command.EXIT, 1);
        yahooCommand = new Command("YAHOO", Command.SCREEN, 2);
        zindellCommand = new Command("ZINDELL", Command.SCREEN, 3);
        form.addCommand(exitCommand);
        form.addCommand(yahooCommand);
        form.addCommand(zindellCommand);
    }

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

    public void pauseApp()
    {
    }

    public void destroyApp(boolean condition)
    {
        display = null;
        form = null;
    }
}

```

The MIDP's UI doesn't have a command layout manager to manage how the commands are displayed on the device screen. The device decides how Commands are mapped to soft-buttons based on the types, priorities and the number of

commands. If the midlet has several commands, you better add the command, you use the most, first and give it the highest priority.

The `CommandListener` is the interface that dictates the `Command` event processing method that should be declared in the class from which the event listener is instantiated. The `CommandListener` interface provide the following method:

```
public void commandAction(Command command, Displayable displayable)
```

The displayable is the event source and the command is the command that was invoked. Every class can implement the `CommandListener` interface and once this class is instantiated the new instance can listen to multiple `Displayable` objects. However, a `Displayable` object can have only one registered `CommandListener` object.

Registering the `CommandListener` object as the listener of a `Displayable` object can be done using the following method, that was declared in the `Displayable` class:

```
public void setCommandListener(CommandListener listener)
```

The following midlet presents a simple events handling case, in which the user can choose one of the two commands: "National" or "Finance" and watch how the presented ticker is changed respectively.

```
//filename:CurrentNews.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;
```

```
public class CurrentNews extends MIDlet implements CommandListener
```

```

{
    private Display display;
    private Form form;
    private Command exitCommand, nationalCommand, financeCommand;

    public CurrentNews()
    {
        display = Display.getDisplay(this);
        form = new Form("** N E W S **");
        exitCommand = new Command("Exit", Command.EXIT, 1);
        nationalCommand = new Command("National", Command.SCREEN, 4);
        financeCommand = new Command("Finance", Command.SCREEN, 5);
        form.setCommandListener(this);
        form.setCommandListener(this);
        form.setCommandListener(this);
        form.addCommand(exitCommand);
        form.addCommand(nationalCommand);
        form.addCommand(financeCommand);
    }

    public void commandAction(Command command, Displayable displayable)
    {
        if(command==exitCommand)
        {
            destroyApp(true);
            notifyDestroyed();
        }
        else
        {
            if(command==nationalCommand)
            {
                form.setTicker(new Ticker("Final peace agreement was
signed between the Palestinians and the Israelis"));
            }
            else
            {
                form.setTicker(new Ticker("The NIS-$ exchange rate is
1.2"));
            }
        }
    }

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

```

```

    public void pauseApp()
    {
    }

    public void destroyApp(boolean condition)
    {
        display = null;
        form = null;
        exitCommand = null;
        nationalCommand = null;
        financeCommand = null;
    }
}

```

The events handling can be done using inner classes just like when using the J2SE.

The following example presents this way of action.

//filename:CurrentNews.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
 //All rights reserved. No part of the contents of this program may be
 //reproduced or transmitted in any form or by any means without the
 //written permission of the publisher.

```

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

```

```

public class CurrentNews extends MIDlet
{
    private Display display;
    private Form form;
    private Command exitCommand, nationalCommand, financeCommand;

    public CurrentNews()
    {
        display = Display.getDisplay(this);
        form = new Form("*** N E W S ***");
        exitCommand = new Command("Exit", Command.EXIT, 1);
        nationalCommand = new Command("National", Command.SCREEN, 4);
        financeCommand = new Command("Finance", Command.SCREEN, 5);
    }
}

```

```

        CommandListener listener = new CommandHandler();
        form.setCommandListener(listener);
        form.setCommandListener(listener);
        form.setCommandListener(listener);
        form.addCommand(exitCommand);
        form.addCommand(nationalCommand);
        form.addCommand(financeCommand);
    }

```

```

class CommandHandler implements CommandListener
{
    public void commandAction(Command command, Displayable
displayable)
    {
        if(command==exitCommand)
        {
            destroyApp(true);
            notifyDestroyed();
        }
        else
        {
            if(command==nationalCommand)
            {
                form.setTicker(new Ticker("Final peace
aggreement was signed between the Palestinians and the Israelis"));
            }
            else
            {
                form.setTicker(new Ticker("The NIS-$ exchange
rate is 1.2"));
            }
        }
    }
}

public void startApp()
{
    System.out.println("startApp()");
    display.setCurrent(form);
}

public void pauseApp()
{
    System.out.println("pauseApp()");
}

```



```

    }

    public void destroyApp(boolean condition)
    {
        System.out.println("destroyApp()");
        display = null;
        form = null;
        exitCommand = null;
        nationalCommand = null;
        financeCommand = null;
    }
}

```

One of the biggest differences between the J2ME and the J2SE is the ability to deliver the event to multiple event listeners. This ability exists in the J2SE and doesn't exist in the J2ME (the J2ME does support this ability when dealing with records listeners).

Calls to event handling methods are made on the same thread where the event occurs. Therefore, the event handling methods should return immediately, otherwise the midlets will be blocked. When the event handling methods don't return immediately, the solution might be using another thread. The following example presents a midlet that starts a new thread when the event handling method is invoked. The midlet computes the factorial of the number the user enters. In order to make the factorial computation longer than usual, the `Thread.sleep()` method is invoked.

//filename:FactorialComputer.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
 //All rights reserved. No part of the contents of this program may be
 //reproduced or transmitted in any form or by any means without the
 //written permission of the publisher.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
```

```
public class FactorialComputer extends MIDlet
{
    private Display display;
    private Form firstForm;
    private Form secondForm;
    private Command restartCommand, exitCommand, computeCommand;
    private TextField tf;

    public FactorialComputer()
    {
        display = Display.getDisplay(this);

        //creating two forms
        firstForm = new Form("Factorial Computer");
        secondForm = new Form("Result");

        //creating the commands
        exitCommand = new Command("Exit", Command.EXIT, 1);
        computeCommand = new Command("Compute", Command.OK, 1);
        restartCommand = new Command("Restart", Command.OK, 1);

        //creating a command listener
        CommandListener listener = new CommandHandler();

        //creating a text field
        tf = new TextField("The number you want to compute its factorial ",
                           "", 30, TextField.NUMERIC);

        //setting command listeners to the two forms
        firstForm.setCommandListener(listener);
        secondForm.setCommandListener(listener);

        //adding the compute and the exit commands to the first form
        firstForm.addCommand(computeCommand);
        firstForm.addCommand(exitCommand);

        //adding the text field to the first form
        firstForm.append(tf);
    }

    class Computer implements Runnable
    {

```

```

private long number;

Computer(int number)
{
    this.number=number;
}

public void run()
{
    long result=1;
    for(int i=1; i<=number; i++)
    {
        result*=i;
        Gauge g = (Gauge)secondForm.get(0);

        //setting the gauge a new value
        g.setValue(i);
        try
        {
            Thread.sleep(50);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }

    //removing the gauge from the secondForm
    secondForm.delete(0);

    //creating a string item
    StringItem si = new StringItem("The factorial of "
        + number + " is ", String.valueOf(result));

    //adding the string item to secondForm
    secondForm.append(si);

    //adding the restart command to the secondForm
    secondForm.addCommand(restartCommand);
}

}

class CommandHandler implements CommandListener
{
    private Thread thread = null;

```

```

        public void commandAction(Command command, Displayable
displayable)
        {
            if(displayable==firstForm) //the user see the first form
            {
                if(command==computeCommand) //the first form is the
current one
                {
                    int number =
Integer.parseInt(((TextField)firstForm.get(0)).getString());
                    secondForm.append(new Gauge("computing...",
false, number, 1));

                    display.setCurrent(secondForm);
                    thread = new Thread(new Computer(number));
                    thread.start();
                }
                else
                {
                    if(command==exitCommand)
                    {
                        destroyApp(true);
                        notifyDestroyed();
                    }
                }
            }
            else
            {
                if(displayable==secondForm) //the second form is the
current one
                {
                    //removing the string item from the second form
                    secondForm.delete(0);
                    secondForm.removeCommand(restartCommand);

                    //setting the first form as the current form
                    display.setCurrent(firstForm);
                }
            }
        }
    }

    public void startApp()
    {
        System.out.println("startApp()");
    }

```

```

        //setting the first form as the current form
        display.setCurrent(firstForm);
    }

    public void pauseApp()
    {
        System.out.println("pauseApp()");
    }

    public void destroyApp(boolean condition)
    {
        System.out.println("destroyApp()");

        //setting nulls in the diferent variables
        display = null;
        firstForm = null;
        secondForm = null;
        computeCommand = null;
        restartCommand = null;
        exitCommand = null;
        tf = null;
    }
}

```

List

The List class extends the Screen class and implements the Choice interface. When a List object is displayed, the user sees a group of choices he can to choose from.

The Choice interface has the following methods:

```
public int append(String str, Image img)
```

```
public void detele(int elementNum)
```

```
public Image getImage(int elementNum)
```

```
public int getSelectedFlags(boolean []vec)
```

```
public int getSelectedIndex()
```

```
public String getString(int elementNum)
```

```
public void insert(int elementNum, String str, Image imgPart)
```

```
public boolean isSelected(int elementNum)
```

```
public void set(int elementNum, String strPart, Image imgPart)
```

```
public void setSelectedFlags(boolean []vec)
```

```
public void setSelectedIndex(int elementNum, boolean selected)
```

```
public void size()
```

The List class has the following constructors:

```
List(String title, int listType)
```

```
List(String title, int listType, String []vec, Image []img)
```

Three types of Choice objects exist:

Exclusive-Choice:

When exactly one element must be selected at any given time (unless the Choice doesn't have elements)

Implicit-Choice (only a list can be of this type):

The same as Exclusive-Choice and in addition, the focused element is implicitly the selected one when a Command is initiated.

Multiple-Choice:

Any number of elements can be chosen.

The different three types are differentiating by their appearance. The following midlet presents these three types of Choice objects. The following midlet presents three different list objects (Exclusive, Implicit and Multiple).

```
//filename:ListTypes.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;
```

```
public class ListTypes extends MIDlet  
{  
    private Display display;  
    private List lists[];  
    private String vec[];  
    private Command nextCommand, exitCommand;  
    private int counter = 0;  
  
    public ListTypes()
```

```

{
    //create 2 implicit commands
    nextCommand = new Command("Next", Command.SCREEN, 1);
    exitCommand = new Command("Exit", Command.EXIT, 1);

    //create an array of 4 options
    vec = new String[4];
    vec[0] = "ZINDELL";
    vec[1] = "JACADO";
    vec[2] = "ZOMPIX";
    vec[3] = "Y2KMULTIMEDIA";

    //create 3 implicit lists
    lists = new List[3];
    lists[0] = new List("IMPLICIT", List.IMPLICIT, vec, null);
    lists[1] = new List("EXCLUSIVE", List.EXCLUSIVE, vec, null);
    lists[2] = new List("MULTIPLE", List.MULTIPLE, vec, null);

    //instantiating the listener
    CommandListener listener = new CommandListener()
    {
        public void commandAction(Command command,
Displayable displayable)
        {
            if(command==nextCommand)
            {
                if(counter==lists.length-1)
                {
                    counter=-1;
                }
                counter++;
                display.setCurrent(lists[counter]);
            }
            else
            {
                if(command==exitCommand)
                {
                    destroyApp(true);
                    notifyDestroyed();
                }
            }
        }
    }
};

```



```

        //add the 2 implicit commands to each one
        //of the lists and set to each one of them
        //the event listener
        for(int i=0; i<lists.length; i++)
        {
            lists[i].addCommand(nextCommand);
            lists[i].addCommand(exitCommand);
            lists[i].setCommandListener(listener);
        }

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void startApp()
    {
        display.setCurrent(lists[counter]);
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        for(int i=0; i<lists.length; i++)
        {
            lists[i] = null;
        }
        display = null;
        lists = null;
        nextCommand = null;
        exitCommand = null;
    }
}

```

The number of elements the Choice object has (in this case, the choice object is a List object) can be retrieved using the size() method. The elements within the Choice object are referred to by their indexes (starting at 0). An image, which is associated with an item, must be immutable. The following middle displays a list with images near each one of its elements.

```
//filename:ListOfImages.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;
```

```
public class ListOfImages extends MIDlet  
{  
    private Display display;  
    private List list;  
    private String vec[];  
    private Image images[];  
    private Command exitCommand;  
  
    public ListOfImages()  
    {  
        //create an implicit command  
        exitCommand = new Command("Exit", Command.EXIT, 1);  
  
        //create an array of 3 images  
        images = new Image[3];  
        try  
        {  
            images[0] = Image.createImage("/rectangle.png");  
            images[1] = Image.createImage("/circle.png");  
            images[2] = Image.createImage("/triangle.png");  
        }  
        catch(Exception e)  
        {  
            e.printStackTrace();  
        }  
  
        //create an array of three strings  
        vec = new String[3];  
        vec[0] = "Rectangle";  
        vec[1] = "Circle";  
        vec[2] = "Triangle";  
    }  
}
```

```

//Creating the list
list = new List("My Nice List", List.IMPLICIT, vec, images);

//instantiating the listener
CommandListener listener = new CommandListener()
{
    public void commandAction(Command command,
Displayable displayable)
    {
        if(command==exitCommand)
        {
            destroyApp(true);
            notifyDestroyed();
        }
    }
}

;

//add the implicit command to the list
list.addCommand(exitCommand);
list.setCommandListener(listener);

//getting the display object
display = Display.getDisplay(this);

}

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

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    display = null;
    list = null;
    exitCommand = null;
    for(int i=0; i<vec.length; i++)
    {
        vec[i] = null;
    }
}

```

```

        images[i] = null;
    }
    vec = null;
    images = null;
}
}

```

When the user change his\her selected item from an **implicit** choice List the CommandListener object, which was registered with that list, is notified (the commandAction method is invoked on it and the List.SELECT_COMMAND - a command - is sent as a parameter). The following midlet has an implicit list from which the user can choose one item. The midlet responds to any change in the selected item. Lists of the other types also respond to any change in the selected item.

//filename:ListDemo.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
 //All rights reserved. No part of the contents of this program may be
 //reproduced or transmitted in any form or by any means without the
 //written permission of the publisher.

```

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;

```

```

public class ListDemo extends MIDlet
{
    private Display display;
    private List list;
    private String vec[];
    private Ticker tickers[];
    private Command exitCommand;

    public ListDemo()
    {
        //create implicit command
        exitCommand = new Command("Exit", Command.EXIT, 1);
    }
}

```

```

//create an array of 4 options
vec = new String[4];
vec[0] = "ZINDELL";
vec[1] = "JACADO";
vec[2] = "ZOMPIX";
vec[3] = "Y2KMULTIMEDIA";

//create an array of 4 tickers
tickers = new Ticker[4];
tickers[0] = new Ticker("http://www.zindell.com");
tickers[1] = new Ticker("http://www.jacado.com");
tickers[2] = new Ticker("http://www.zompix.com");
tickers[3] = new Ticker("http://www.y2kmultimedia.com");

//create implicit list
list = new List("Awesome Links...", List.IMPLICIT, vec, null);

//instantiating the listener
CommandListener listener = new CommandListener()
{
    public void commandAction(Command command,
Displayable displayable)
    {
        if(displayable==list &&
command==List.SELECT_COMMAND)
        {
            int index =
list.getSelectedIndex();

            System.out.println(vec[index]
                                + " is
selected");

            list.setTicker(tickers[index]);
        }
        else
        {
            if(command==exitCommand)
            {
                destroyApp(true);
                notifyDestroyed();
            }
        }
    }
}

```

```

        ;

        //add the implicit command to the list
        //and set a listener
        list.addCommand(exitCommand);
        list.setCommandListener(listener);

        //getting the display object
        display = Display.getDisplay(this);
    }

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

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        list = null;
        exitCommand = null;
    }
}

```

It is possible constructing menus by associating logical commands with elements in an implicit list. When doing so, there is no need to instantiate the `Command` class and attach the commands to the list. All you have to do is registering a `CommandListener` to the list (as was done in the last example).

The following example presents the difference between the three types of lists.

//filename:ListDemo.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.

//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;
```

```
public class ListDemo extends MIDlet  
{  
    private Display display;  
    private List mainList, implicitList, exclusiveList, multipleList;  
    private String vec[];  
    private Ticker tickers[], moreThanOne;  
    private Command exitCommand;  
    private Command okCommand;  
    private Command returnCommand;  
  
    public ListDemo()  
    {  
        //create an array of 4 options  
        vec = new String[4];  
        vec[0] = "Zindell";  
        vec[1] = "Jacado";  
        vec[2] = "Zompix";  
        vec[3] = "Y2Kmultimedia";  
        System.out.println("vec array was created");  
  
        //create an array of 3 tickers  
        tickers = new Ticker[4];  
        tickers[0] = new Ticker("zindell zindell zindell");  
        tickers[1] = new Ticker("jacado jacado jacado");  
        tickers[2] = new Ticker("zompix zompix zompix");  
        tickers[3] = new Ticker("Y2Kmultimedia Y2Kmultimedia  
Y2Kmultimedia");  
        System.out.println("tickers array was created");  
  
        //create the moreThanOne ticker  
        moreThanOne = new Ticker("choose just one option!");  
  
        //create the main list  
        mainList = new List("List Types", List.IMPLICIT);  
        mainList.append("Implicit List",null);  
        mainList.append("Exclusive List",null);  
    }  
}
```

```

mainList.append("Multiple List",null);
mainList.append("Exit",null);
System.out.println("main list was created");

//create the other three lists
implicitList = new List("Implicit List", List.IMPLICIT, vec, null);
exclusiveList = new List("Exclusive List", List.EXCLUSIVE, vec, null);
multipleList = new List("Multiple List", List.MULTIPLE, vec, null);
returnCommand = new Command("Return", Command.BACK, 1);
okCommand = new Command("Ok", Command.OK, 1);
exitCommand = new Command("Exit", Command.EXIT, 1);
implicitList.addCommand(returnCommand);
exclusiveList.addCommand(returnCommand);
multipleList.addCommand(returnCommand);
multipleList.addCommand(okCommand);
exclusiveList.addCommand(okCommand);
System.out.println("the three different lists were created");

//instantiating the listener
CommandListener listener = new CommandListener()
{

    public void changeTicker(Displayable displayable)
    {
        List currentList = (List)displayable;
        int temp = currentList.getSelectedIndex();
        if(temp!=-1) //not a multiple list
        {
            System.out.println("within the changeTicker
method. temp!=-1");

            switch(temp)
            {
                case 0:
                case 1:
                case 2:
                case 3:

                    currentList.setTicker(tickers[temp]);
            }
        }
        else //a multiple list
        {
            System.out.println("within the changeTicker
method. temp=-1");

            boolean vec[] = new
boolean[currentList.size()];

```



```

        currentList.getSelectedFlags(vec);
        int counter=0, chosen=0;
        for(int i=0; i<vec.length; i++)
        {
            if(vec[i])
            {
                chosen = i;
                counter++;
            }
        }
        if(counter==1)
        {
            System.out.println("counter==1");

currentList.setTicker(tickers[chosen]);
        }
        else
        {
            System.out.println("counter!=1");

currentList.setTicker(moreThanOne);
        }
    }

    public void exit()
    {
        destroyApp(true);
        notifyDestroyed();
    }

    public void commandAction(Command command,
Displayable displayable)
    {
        System.out.println("inside commandAction");
        if(displayable==mainList &&
command==List.SELECT_COMMAND)
        {
            switch(mainList.getSelectedIndex())
            {
                case 0:

display.setCurrent(implicitList);

                break;
                case 1:

```



```

        {
            if(displayable==multipleList)
            {

if(command==okCommand)

                                {

changeTicker(displayable);

                                }
                                else
                                {

if(command==returnCommand)

                                                {

display.setCurrent(mainList);

                                                }

                                }

                                }

                                }

                                }

                                }

                                }

                                }

;
System.out.println("the listener was instantiated");

//set a listener to each one of the lists
mainList.setCommandListener(listener);
implicitList.setCommandListener(listener);
exclusiveList.setCommandListener(listener);
multipleList.setCommandListener(listener);

//getting the display object
display = Display.getDisplay(this);
}

public void startApp()
{
    mainList.setSelectedIndex(0,true);
    display.setCurrent(mainList);
}

public void pauseApp()
{

```

```

    }

    public void destroyApp(boolean cond)
    {
        display = null;
        mainList = null;
        multipleList = null;
        implicitList = null;
        exclusiveList = null;
        exitCommand = null;
    }
}

```

TextBox

The TextBox class extends the Screen class. A TextBox object allows the user to enter and edit text. The TextBox object has a maximum stored capacity. The number of characters that may be displayed at any time can't be higher than it. When the TextBox object is constructed the maximum stored capacity number is set. The device determines the arrangement of the displayed characters. The device can impose a limit to the number of characters the TextBox can display. The device limit can be retrieved using the getMaxSize() method.

Once the TextBox is instantiated the data that can be entered into it is constrained according to the value (one of the following) that was set in the TextBox instantiation:

Name	Value	Meaning
TextField.ANY	0	Any text
TextField.CONSTRAINT_MASK	0xFFFF	This constant is used in order to find the current constraint when the TextField.PASSWORD is used (using

		the & operation between TextFiedl.CONSTRAINT_MAST and the value getConstraints() returns).
TextField.EMAILADDR	1	Only email address
TextField.NUMERIC	2	Only integer value
TextField.PASSWORD	0x10000	This constrained is combined with other constrained using the operator
TextField.PHONENUMBER	3	Only a phone number and several optional non-numerical characters
TextField.URL	4	Only a URL address

These value are final static fields that were declared as part of the TextField class. Using these values enables restricting the user input in a variety of ways.

The basic simple constructor the TextBox class has is:

TextBox(String title, String text, int maxSize, int constraints)

title - the description label of the text box.

text - the default text in the TextBox (the default value of the TextBox)

maxSize - the maximum stored capacity

constraints - one of the final static fields that were declared as part of the TextField class and represent constraints on the text the TextBox can hold.

The following midlet present several text boxes. The difference between each one of them is the constraint that was set.

//filename:TextBoxDemo.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.

//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
```

```
public class TextBoxDemo extends MIDlet
{
    private Display display;
    private List mainList;
    private String vec[];
    private TextBox tBoxes[];
    private TextBox description;
    private Command backCommand, exitCommand, okCommand;

    public TextBoxDemo()
    {
        //create the backCommand, okCommand and the exitCommand
        backCommand = new Command("Back", Command.SCREEN, 1);
        exitCommand = new Command("Exit", Command.EXIT, 1);
        okCommand = new Command("Ok", Command.OK, 1);

        //create the description TextBox
        description = new TextBox("Description", "", 40, TextField.ANY);
        description.addCommand(backCommand);

        //create an array of 6 options
        String vec[] = {"ANY", "EMAILADDR", "NUMERIC",
                        "PHONENUMBER", "URL",
"PASSWORD"};
        System.out.println("vec array was created");

        //create an array of 6 text boxes
        tBoxes = new TextBox[6];
        int constraints[] = {    TextField.ANY, TextField.EMAILADDR,
TextField.NUMERIC,
                                TextField.PHONENUMBER,
TextField.URL, TextField.PASSWORD};
        for(int i=0; i<tBoxes.length; i++)
        {
            tBoxes[i] = new TextBox(vec[i], "", 20, constraints[i]);
            tBoxes[i].addCommand(backCommand);
        }
    }
}
```

```

        tBoxes[i].addCommand(okCommand);
    }
    System.out.println("tBoxes array was created");

    //create the main list
    mainList = new List("TextBox Types", List.IMPLICIT);
    for(int i=0; i<vec.length; i++)
    {
        mainList.append(vec[i],null);
    }
    System.out.println("main list was created");

    //instantiating the listener
    CommandListener listener = new CommandListener()
    {
        TextBox previous;
        public void commandAction(Command command,
Displayable displayable)
        {
            if(command==backCommand)
            {
                if(displayable==description)
                {
                    display.setCurrent(previous);
                }
                else
                {
                    display.setCurrent(mainList);
                }
            }
            else
            {
                if(command==exitCommand)
                {
                    destroyApp(true);
                    notifyDestroyed();
                }
                else
                {
                    if(command==List.SELECT_COMMAND && displayable==mainList)
                    {
                        previous =
tBoxes[((List)displayable).getSelectedIndex()];
                        display.setCurrent(previous);
                    }
                }
            }
        }
    };

```

```

else
{
    if(command==okCommand)
    {

description.setString("text box content is " + previous.getString());

display.setCurrent(description);

    }
}
}
}
}

;
System.out.println("the listener was instantiated");

//set a listener to each one of the lists
mainList.setCommandListener(listener);
for(int i=0; i<tBoxes.length; i++)
{
    tBoxes[i].setCommandListener(listener);
}

//set a listener to the description text box
description.setCommandListener(listener);

//getting the display object
display = Display.getDisplay(this);
}

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

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    display = null;
    mainList = null;
}

```



```

        for(int i=0; i<tBoxes.length; i++)
        {
            tBoxes[i] = null;
        }
        tBoxes = null;
        backCommand = null;
        exitCommand = null;
    }
}

```

Alert

The *Alert* class extends the *Screen* class. An *Alert* object shows the user text and an image and waits for a certain period of time before disappearing. The *Alert* class has two constructors:

`Alert(String theTitle)`

and

`Alert(String theTitle, String alertText, Image alertImage, AlertType alertType)`

The timeout of the alert can be set using the `setTimeout()` method. The timeout can be set to be infinity by using the final static field `Alert.FOREVER`, and sending this value to the `setTimeout()` method. An alert that its timeout is infinity won't disappear unless the user presses the Done command. If the midlet doesn't call the `setTimeout()` method then the timeout the alert shell have is the default timeout value specified by the device. The default timeout value the device has can be retrieved using the `getDefaultTimeout()` method.

When the alert is timed out, the device presents the next displayable. To do so, the alert is set as the current displayable object using the following method (belongs to Display class):

```
public void setCurrent(Alert alert, Displayable nextDisplayable)
```

If the alert is set as the current displayable using the setCurrent method that has one argument:

```
public void setCurrent(Displayable displayable)
```

The setCurrent implementation will do the following: setCurrent(alert, getCurrent());

It is impossible adding a command to an alert. Calling the addCommand() method on a given alert will result in throwing IllegalStateException.

The alert purpose is informing the users about errors and other exceptional situations. Each alert has the alertType attribute that indicate the alert nature.

The alert type can be one of the following:

AlertType.ALARM

AlertType.CONFIRMATION

AlertType.ERROR

AlertType.INFO

AlertType.WARNING

The alert type is set when the Alert class is instantiated.

The following midlet presents a simple alert that the user get when he enters the salary he\she wants to get.

```
//filename:SimpleAlertDemo.java
```

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
```

```
public class SimpleAlertDemo extends MIDlet
{
    private Display display;
    private TextBox firstTB, secondTB;
    private Command backCommand, exitCommand, okCommand;
    private Alert lowAlert, highAlert;

    public SimpleAlertDemo()
    {
        //create the backCommand, okCommand and the exitCommand
        backCommand = new Command("Back", Command.SCREEN, 1);
        exitCommand = new Command("Exit", Command.EXIT, 1);
        okCommand = new Command("Ok", Command.OK, 1);

        //create the description TextBox
        firstTB = new TextBox("What is the salary you want ?", "10000", 40,
TextField.NUMERIC);
        secondTB = new TextBox("Good Luck", "", 60, TextField.ANY);
        firstTB.addCommand(okCommand);
        secondTB.addCommand(backCommand);
        secondTB.addCommand(exitCommand);

        //create the two alerts
        lowAlert = new Alert("That's All !?", "",
                                null, AlertType.WARNING);
        highAlert = new Alert("You ask a lot!", "", null, AlertType.INFO);
        lowAlert.setTimeout(Alert.FOREVER);
        highAlert.setTimeout(Alert.FOREVER);

        //instantiating the listener
        CommandListener listener = new CommandListener()
        {
            public void exit()
```

```

        {
            destroyApp(true);
            notifyDestroyed();
        }

        public void commandAction(Command command,
Displayable displayable)
        {
            if(command==exitCommand)
            {
                exit();
            }
            else
            {
                if(displayable==firstTB)
                {
                    if(command==okCommand)
                    {
                        int sum =
Integer.parseInt(firstTB.getString());
                        secondTB.setString("We
hope the salary of " + sum
+ " shell satisfied you
...");
                        if(sum>10000)
                        {
                            highAlert.setString(sum+" is quite a lot!");
                            display.setCurrent(highAlert, secondTB);
                        }
                        else
                        {
                            lowAlert.setString(sum+" is not a lot!");
                            display.setCurrent(lowAlert, secondTB);
                        }
                    }
                }
            }
            else
            {
                if(displayable==secondTB)
                {
                    if(command==backCommand)

```

```

        {
display.setCurrent(firstTB);
        }
    }
}

;
System.out.println("the listener was instantiated");

//setting the listeners
firstTB.setCommandListener(listener);
secondTB.setCommandListener(listener);

//getting the display object
display = Display.getDisplay(this);
}

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

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    display = null;
    firstTB = null;
    secondTB = null;
    okCommand = null;
    backCommand = null;
    exitCommand = null;
}
}

```

Form and Items

A Form object is a screen (the Form class extends Screen) that represents a surface on which it is possible to put items (images, read-only text fields, editable text fields, editable date fields, gauges, and choice groups).

Items within a Form are referred to by their indexes. The indexes are integers in the range from 0 to `size()-1`, with zero referring to the first item and `size()-1` to the last item. The `size()` method returns the number of items the form has. Each one of the items can be placed within only one form.

When the user modify the state of an interactive item contained within the form, the system notifies the midlet by calling the `itemStateChange()` method of the listener declared to the form (with the `setItemStateListener()` method).

It is possible adding commands to the form, and set a command listener (using the `setCommandListener()` method) to the form (just as we did before with lists, text boxes...).

The Form class has two constructors:

`Form(String title)`

This constructor is used for the creation of an empty form (without any items on it).

`Form(String title, Item vec[])`

This constructor is used for the creation of a form with initial items.

The Form class has different methods that enable the control of the items that were positioned on the form:

```
public int append(Image img)
```

```
public int append(Item item)
```

```
public int append(String str)
```

```
public void delete(int itemNum)
```

```
public Item get(int itemNum)
```

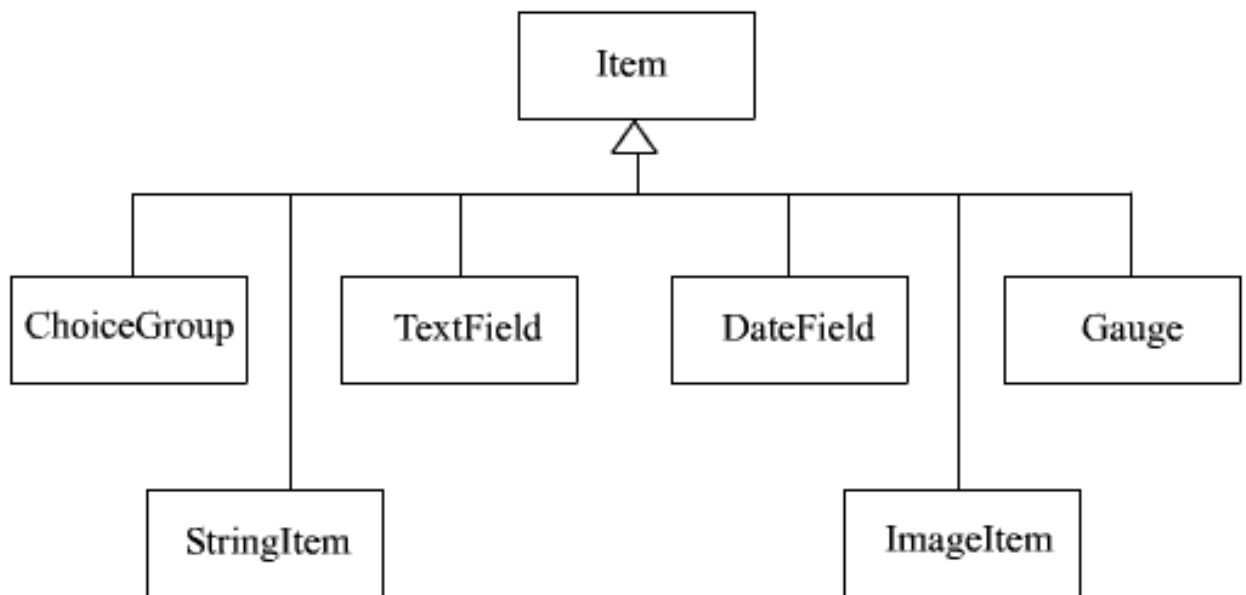
```
public void insert(int itemNum, Item item)
```

```
public void set(int itemNum, Item item)
```

```
public void setItemStateListener(ItemStateListener listener)
```

```
public int size()
```

The Item class is the super class of several other classes that their instances can be used as items within a form. Each item is instantiated from one of the Item subclasses. Each item has a label that most of the devices present near the item itself. The following figure shows the classes that extend the Item class.



The *ChoiceGroup*, *TextField*, *DateField* and *Gauge* items can be interactively changed by the users. The *StringItem* and *ImageItem* items cannot get the focus and therefore, their content can be changed only by the midlet himself and not by the user.

The **StringItem** item contains a label and a string that cannot be changed by the user (only the midlet).

The following midlet presents a simple use of the *StringItem* class.

//filename:StringItemDemo.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
```

```
public class StringItemDemo extends MIDlet implements CommandListener
{
    private Display display;
    private Form form;
    private StringItem stringItem;
    private Command exitCommand;

    public StringItemDemo()
    {
        //create the exitCommand
        exitCommand = new Command("Exit", Command.EXIT, 1);

        //creating the form
        form = new Form("ZINDELL SERVICES");
        form.append(new
StringItem("website:\nwww.zindell.com\n", "call:\n+972+58+355837"));
        form.addCommand(exitCommand);

        //setting the listener
        form.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void exit()
    {
        destroyApp(true);
        notifyDestroyed();
    }

    public void commandAction(Command command, Displayable displayable)
    {
        if(command==exitCommand)
        {
            exit();
        }
    }
}
```

```

        }
    }

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

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        stringItem = null;
        exitCommand = null;
    }
}

```

The **ImageItem** item can contain only an immutable Image object. Each image item has a layout (it can be set in its instantiation) parameter that specifies how it (the image) is aligned on the screen. The layout directives for image items are: LAYOUT_DEFAULT, LAYOUT_LEFT, LAYOUT_RIGHT, LAYOUT_CENTER, LAYOUT_NEWLINE_BEFORE, LAYOUT_NEWLINE_AFTER). The layout directives can be combined with the | operator. However, because of the device constraints such as limited screen size, the layout you set might doesn't have the effect you thought. Using the setAltText() method an alternative text can be set fot an image item so it will be displayed in those cases in which the image can't be displayed.

The following midlet presents a simple use of the ImageItem class.

//filename:ImageItemDemo.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
```

```
public class ImageItemDemo extends MIDlet implements CommandListener
{
    private Display display;
    private Form form;
    private Command exitCommand;
    private Item item;
    private Image img;

    public ImageItemDemo()
    {
        //create the backCommand, okCommand and the exitCommand
        exitCommand = new Command("Exit", Command.EXIT, 1);

        //creating the form
        form = new Form("ZINDELL SERVICES");

        //creating an image item
        try
        {
            img = Image.createImage("/PacManImg.png");
            item = new ImageItem("PAC-MAN",
                                img,
                                ImageItem.LAYOUT_DEFAULT,
                                "[PACMAN]");
        }
        catch(IOException exception)
        {
            exception.printStackTrace();
            item = new StringItem("PAC-MAN", "problem in loading the
picture");
        }
        form.append(item);

        //adding the exit command
        form.addCommand(exitCommand);
    }
}
```

```

        //setting the listener
        form.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void exit()
    {
        destroyApp(true);
        notifyDestroyed();
    }

    public void commandAction(Command command, Displayable displayable)
    {
        if(command==exitCommand)
        {
            exit();
        }
    }

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

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        item = null;
        img = null;
        exitCommand = null;
    }
}

```

The **ChoiceGroup** item is very similt to list. The difference between a choice group item and a list is the implicit choice type that the List class supports and the

ChoiceGroup doesn't. The following midlet present a simple use of the ChoiceGroup class. The following example doesn't present an events handling for the ChoiceGroup item.

```
//filename:ChoiceGroupItemDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;
```

```
public class ChoiceGroupItemDemo extends MIDlet implements CommandListener  
{  
    private Display display;  
    private Form form;  
    private Command exitCommand;  
    private Item item;  
  
    public ChoiceGroupItemDemo()  
    {  
        //create the backCommand, okCommand and the exitCommand  
        exitCommand = new Command("Exit", Command.EXIT, 1);  
  
        //creating the form  
        form = new Form("ZINDELL SERVICES");  
  
        //creating a ChoiceGroup item  
        item = new ChoiceGroup(    "Java Links",  
                                   Choice.EXCLUSIVE,  
                                   new String[] {"zindell", "javasoft",  
"javareport"},  
                                   null);  
        form.append(item);  
  
        //adding the exit command  
        form.addCommand(exitCommand);  
    }  
}
```

```

        //setting the listener
        form.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void exit()
    {
        destroyApp(true);
        notifyDestroyed();
    }

    public void commandAction(Command command, Displayable displayable)
    {
        if(command==exitCommand)
        {
            exit();
        }
    }

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

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        item = null;
        exitCommand = null;
    }
}

```

A **Gauge** item is a bar graph that represents a value in the range between zero to the `maxValue` attribute of the gauge. A gauge object can be constructed using the following constructor:

`Gauge(String label, boolean interactive, int maxValue, int initialValue)`

The Gauge class has the following methods:

```
public int getMaxValue()
```

```
public int getValue()
```

```
public boolean isInteractive()
```

```
public void setMaxValue(int maxValue)
```

```
public void setValue(int value)
```

The gauge can be interactive (the user can change it) or it can be non-interactive (the user can't change it). Usually, these two types of gauges have different appearance. However, the midlet can easily set the value of gauge using the `setValue()` method. The following midlet presents two gauges. One is interactive and one is non-interactive. The user can choose which kind of gauge he/she wants to see.

```
//filename:GaugeDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;
```

```
public class GaugeDemo extends MIDlet  
{
```

```
    private Display display;  
    private List menu;  
    private Form form;  
    private Command backCommand, exitCommand;
```

```
    public GaugeDemo()  
    {
```

```
        //create the backCommand, okCommand and the exitCommand  
        backCommand = new Command("Back", Command.SCREEN, 1);
```

```

exitCommand = new Command("Exit", Command.EXIT, 1);

//creating the menu
String choices[] = {"Interactive","Non-Interactive"};
menu = new List("GAUGE TYPES", List.IMPLICIT, choices, null);
menu.addCommand(exitCommand);

//creating the form
form = new Form("GAUGE in ACTION");
form.addCommand(backCommand);

//instantiating the listener
CommandListener listener = new CommandListener()
{
    public void exit()
    {
        destroyApp(true);
        notifyDestroyed();
    }

    public void commandAction(Command command,
Displayable displayable)
    {
        if(displayable==menu)
        {
            if(command==exitCommand)
            {
                exit();
            }
            else
            {
                if(command==List.SELECT_COMMAND)
                {
                    int chosen =
menu.getSelectedIndex();

                    if(form.size()==1)
                    {
                        form.delete(0);
                    }
                    if(chosen==0)
                    {
                        form.append(new
Gauge("Interactive",true, 10, 3));

                        display.setCurrent(form);
                    }
                }
            }
        }
    }
}

```



```

        }
        else
        {
            if(chosen==1)
            {

form.append(new Gauge("Non-Interactive",false, 10, 3));

display.setCurrent(form);

            }
        }
    }
}
else
{
    if(command==backCommand)
    {
        display.setCurrent(menu);
    }
}
}

;

//setting the listeners
menu.setCommandListener(listener);
form.setCommandListener(listener);

//getting the display object
display = Display.getDisplay(this);
}

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

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    display = null;

```

```

        menu = null;
        form = null;
        backCommand = null;
        exitCommand = null;
    }
}

```

A **DateField** item is used for presenting date and time. The date field item is editable. The DateField has the following constructors:

```
public DateField(String label, int mode)
```

and

```
public DateField(String label, int mode, TimeZone zone)
```

The date item can have one of the following three input modes:

DATE - The user can set only date information. When using this mode, the time should be set to 0.

TIME - The user can set only time information (hours and minutes). When using this mode, the date should be set to 0 (1/1/1970).

DATE_TIME - The user can set both the clock time and the date values.

The following midlet presents a simple use of the DateField class.

```
//filename:DateFieldDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import java.util.*;
```

```

public class DateFieldDemo extends MIDlet implements CommandListener
{
    private Display display;
    private Form form;
    private List menuList;
    private Command exitCommand, backCommand;
    private DateField item;
    private String modes[] = {"DATE","TIME","DATE_TIME"};

    private static final long MILLIS_IN_DAY = 24*60*60*1000;

    public DateFieldDemo()
    {
        //create the backCommand and the exitCommand
        exitCommand = new Command("Exit", Command.EXIT, 1);
        backCommand = new Command("Back", Command.BACK, 1);

        //create the menuList
        menuList = new List( "DateField-Modes",
                            List.IMPLICIT,
                            modes,
                            null);
        menuList.addCommand(exitCommand);

        //creating the form
        form = new Form("DateField input mode");
        form.addCommand(backCommand);

        //setting the listener
        form.setCommandListener(this);
        menuList.setCommandListener(this);

        //creating the date field
        item = new DateField("DATE MODE",1);
        form.append(item);

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void exit()
    {
        destroyApp(true);
        notifyDestroyed();
    }
}

```

```

public void commandAction(Command command, Displayable displayable)
{
    if(displayable==menuList)
    {
        if(command==exitCommand)
        {
            exit();
        }
        else
        {
            if(command==List.SELECT_COMMAND)
            {
                int index = menuList.getSelectedIndex();
                item.setInputMode(index+1);
                item.setLabel(modes[index]);
                long ms = System.currentTimeMillis();
                long time = ms % MILLIS_IN_DAY;
                long date = ms - time;
                switch(index)
                {
                    case 0: //DATE
                        item.setDate(new Date(date));
                        break;
                    case 1: //TIME
                        item.setDate(new Date(time));
                        break;
                    case 2: //DATE_TIME
                        item.setDate(new Date(date+time));
                        break;
                }
                display.setCurrent(form);
            }
        }
    }
    else
    {
        if(displayable==form)
        {
            if(command==backCommand)
            {
                display.setCurrent(menuList);
            }
        }
    }
}

```

```

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

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        item = null;
        exitCommand = null;
        menuList = null;
    }
}

```

A **TextField** item contains a text that can be edited interactively by the user. The TextField class has the following constructor:

```
TextField(String label, String text, int maxSize, int constraints)
```

As can be seen from the TextField constructor, the TextBox and the TextField are very similar. Both the TextField and the TextBox contain a text string and a label.

Both the TextField and the TextBox have a maximum number of characters that the user can input. Both the TextField and the TextBox share the concept of input constraints. Even the methods that were declared in these two classes are the same. The difference between the two is the way each one of them is handled.

Since the TextBox class extends Screen, a text box can be treated as a displayable object and displayed directly on the screen. The TextField class extends Item, and therefore a text field can be added as an item to a given form (the Form class extends Screen and therefore a form can be displayed directly on the screen).

The following midlet presents a simple use of the TextField class.

```
//filename:TextFieldDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;
```

```
public class TextFieldDemo extends MIDlet implements CommandListener  
{  
    private Display display;  
    private Form form;  
    private TextField tf;  
    private Command exitCommand;  
  
    public TextFieldDemo()  
    {  
        //create the backCommand, okCommand and the exitCommand  
        exitCommand = new Command("Exit", Command.EXIT, 1);  
  
        //creating the form  
        form = new Form("EMAIL MANAGER");  
        form.append(new TextField("Your email  
is", "haim_michael@zindell.com", 40, TextField.EMAILADDR));  
        form.addCommand(exitCommand);  
  
        //setting the listener  
        form.setCommandListener(this);  
  
        //getting the display object  
        display = Display.getDisplay(this);  
    }  
  
    public void exit()  
    {  
        destroyApp(true);  
        notifyDestroyed();  
    }  
  
    public void commandAction(Command command, Displayable displayable)
```

```

        {
            if(command==exitCommand)
            {
                exit();
            }
        }

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

        public void pauseApp()
        {
        }

        public void destroyApp(boolean cond)
        {
            display = null;
            form = null;
            tf = null;
            exitCommand = null;
        }
    }
}

```

The layout management of the items that you add to a form is done by the MIDP implementation. Each device might layout the items in a different way. The following midlet presents an example of a form with more than one item within it.

//filename:LayoutDemo.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
 //All rights reserved. No part of the contents of this program may be
 //reproduced or transmitted in any form or by any means without the
 //written permission of the publisher.

```

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import java.util.*;

```

```

public class LayoutDemo extends MIDlet implements CommandListener
{
    private Display display;
    private Form form;
    private Item items[];
    private Command exitCommand;

    public LayoutDemo()
    {
        //create the backCommand, okCommand and the exitCommand
        exitCommand = new Command("Exit", Command.EXIT, 1);

        //creating the form
        form = new Form("LAYOUT DEMO");
        form.addCommand(exitCommand);

        //setting the listener
        form.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);

        //creating the items
        items = new Item[6];
        items[0] = new StringItem("StringItem", "This is a string item");
        try
        {
            items[1] = new ImageItem( "ImageItem",
                                     Image.createImage("/pac.png"),
                                     ImageItem.LAYOUT_CENTER,
                                     "[PACMAN]");
        }
        catch(IOException e)
        {
            e.printStackTrace();
            items[1] = new StringItem("The ImageItem wasn't
instantiated", "IOException has happened");
        }
        items[2] = new ChoiceGroup( "ChoiceGroup",
                                   Choice.MULTIPLE,
                                   new String[]
{"ZINDELL", "JACADO"},
                                   null);
        items[3] = new Gauge("Gauge", true, 100, 30);
        items[4] = new DateField("DateField", DateField.DATE);
    }
}

```



```

        ((DateField)items[4]).setDate(new Date());

        items[5] = new
TextField("TextField","BLABLABLA...",20,TextField.ANY);

        //appending the items to the form
        for(int i=0; i<items.length; i++)
        {
            form.append(items[i]);
        }
    }

    public void exit()
    {
        destroyApp(true);
        notifyDestroyed();
    }

    public void commandAction(Command command, Displayable displayable)
    {
        if(command==exitCommand)
        {
            exit();
        }
    }

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

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        for(int i=0; i<items.length; i++)
        {
            items[i] = null;
        }
        items = null;
        exitCommand = null;
    }

```

}

ItemStateChanged Events

When the user change the state of an item (editing a text field, selecting\unselecting elements of a choice group etc...) an ItemStateChanged event is generated. The ItemStateChanged event isn't generated when the application change the item state.

The ItemStateChanged events have a corresponding ItemStateListener interface. The ItemStateListener object is registered to a form using the setItemStateListener() method. A Form object can have only one ItemStateListener object and an ItemStateListener object can listen to more than one form.

The ItemStateListener has the following method:

```
public void itemStateChanged(Item item)
```

When the internal state of an item is changes by the user (the user changes the selected value of a ChoiceGroup item, the user adjusts the value of an interactive Gauge item, the user modifies the value in a TextField\DateField item) this method will be called. The exact moment in which the item internal state is considered as a

new state might be different in different devices. Even though the `itemStateChanged()` method doesn't receive information about the displayable(the form) the item belongs to, this info can be easily retrieved by the midlet itself. (An item cannot be appended to more than one form).

The following midlet present a simple use of the `ItemStateChanged` events.

```
//filename:ItemStateChangedDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;  
import java.util.*;
```

```
public class ItemStateChangedDemo extends MIDlet implements CommandListener,  
ItemStateListener  
{  
    private Display display;  
    private Form form;  
    private Item items[];  
    private Command exitCommand;  
  
    public ItemStateChangedDemo()  
    {  
        //create the backCommand, okCommand and the exitCommand  
        exitCommand = new Command("Exit", Command.EXIT, 1);  
  
        //creating the form  
        form = new Form("LAYOUT DEMO");  
        form.addCommand(exitCommand);  
  
        //setting the listeners  
        form.setCommandListener(this);  
        form.setItemStateListener(this);  
    }  
}
```

```

//getting the display object
display = Display.getDisplay(this);

//creating the items
items = new Item[6];
items[0] = new StringItem("StringItem","This is a string item");
try
{
    items[1] = new ImageItem( "ImageItem",
                             Image.createImage("/pac.png"),
                             ImageItem.LAYOUT_CENTER,
                             "[PACMAN]");
}
catch(IOException e)
{
    e.printStackTrace();
    items[1] = new StringItem("The ImageItem wasn't
instantiated","IOException has happened");
}
items[2] = new ChoiceGroup(      "ChoiceGroup",
                                Choice.MULTIPLE,
                                new String[]
{"ZINDELL","JACADO"},
                                null);
items[3] = new Gauge("Gauge", true, 100,30);
items[4] = new DateField("DateField", DateField.DATE);
((DateField)items[4]).setDate(new Date());

items[5] = new
TextField("TextField","BLABLABLA...",20,TextField.ANY);

//appending the items to the form
for(int i=0; i<items.length; i++)
{
    form.append(items[i]);
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void commandAction(Command command, Displayable displayable)

```

```

    {
        if(command==exitCommand)
        {
            exit();
        }
    }

    public void itemStateChanged(Item item)
    {
        String str = item.getLabel();
        form.setTitle(str);
    }

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

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        for(int i=0; i<items.length; i++)
        {
            items[i] = null;
        }
        items = null;
        exitCommand = null;
    }
}

```

Chapter 4:

Persistent Storage

- • [Introduction](#)
- • [The RecordStore class](#)
- • [Records Handling](#)
- • [Record Events Handling](#)
- • [The RecordEnumeration Interface](#)
- • [Simulating a Multiple Columns Table](#)
- • [The RecordFilter and the RecordComparator](#)

Introduction

The MID profile defines a simple database system, the **RMS** (Record Management System). The RMS holds **record stores** (tables). Each record store (a table) holds **records**, which are the entries (rows) of the record store.

The APIs for managing the RMS are described in the javax.microedition.rms package.

The javax.microedition.rms package includes the following classes:

RecordStore

RecordEnumeration

The javax.microedition.rms package includes the following four interfaces:

RecordListener

RecordFilter

RecordComparator

RecordEnumeration (implementation of MIDP must include an implementation of this interface)

The javax.microedition.rms package also includes the following five exception classes:

InvalidRecordIDException

RecordStoreException

RecordStoreFullException

RecordStoreNotOpenedException

RecordStoreNotFoundException

The RecordStore class

The record store consists of a collection of records, which remain persistent across multiple invocations of the MIDlet. A RecordStore object represents a record store. The device MIDP implementation usually handles a record store as a flat file.

The naming space for record stores (each record store has a name) is common for all the midlets that belong to one MIDlet suite which means that each record store within a MIDlet suite must have a different name.

When a MIDlet suite is removed from the device all its record stores are removed as well. While midlets within the same midlet suite can access each other's record stores, they cannot access record stores that belong to other MIDlet suites.

The record store names must follow the following **rules**:

They are case sensitive

They might consist of any combination of up to 32 unicode characters

They must be unique within the scope of the MIDlet suite

All of the individual record store operations are atomic and synchronous so no corruption can happen with multiple accesses unless the accesses are done from

different threads. In this case it is the midlet responsibility to coordinate between the different accesses.

Each record within a given record store is uniquely identified by its record id (an integer value) that is also used as the primary key for the record. The first record in a record store receives a record id equals to 1. Each subsequent record added to the record store shall have a record id 1 greater than the last record that was added before. Therefore, if two records are added to the record store and the first's record id is n then the second's record id will be $n+1$.

Each record store has a date\time stamp (long integer number) that equals to the last time it was modified and a version (integer value) that is incremented with each modification.

The RecordStore class provides several methods for creating\opening\closing and deleting a record store:

public static RecordStore openRecordStore(String recordStoreName, boolean createIfNecessary)

If there is already a record store with this name in the MIDlet suite to which the midlet belongs to then a reference to the already existing record store will be returned. If the createIfNecessary is true then the record store will be created only if there is a need to do so. Using this method it is possible checking whether a specific record store exists. By sending false as the second argument, if the record store doesn't exist then a RecordNotFoundException will be thrown.

public void closeRecordStore()

This method should be called the same number of times as the openRecordStore() is called in order to close the record store. As long as the record store is open, it

consumes resources. Therefore, each thread that opens a record store should also call the close method explicitly.

public static void deleteRecordStore(String recordStoreName)

If the record store you want to delete is opened or is not exist then a `RecordStoreException` will be thrown.

The following midlet presents a simple record store creating and opening.

```
//filename:RecordStoreCreationDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import javax.microedition.rms.*;
```

```
public class RecordStoreCreationDemo extends MIDlet implements CommandListener  
{  
    private Display display;  
    private Form form;  
    private Item item;  
    private Command exitCommand;  
  
    public RecordStoreCreationDemo()  
    {  
        //creating the exitCommand  
        exitCommand = new Command("Exit", Command.EXIT, 1);  
  
        //creating the form  
        form = new Form("RecordStore creation");  
        form.addCommand(exitCommand);  
  
        //setting the listener  
        form.setCommandListener(this);  
    }  
}
```

```

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void exit()
    {
        destroyApp(true);
        notifyDestroyed();
    }

    public void commandAction(Command command, Displayable displayable)
    {
        if(command==exitCommand)
        {
            exit();
        }
    }

    public void startApp()
    {
        RecordStore rs = null;
        try
        {
            rs = RecordStore.openRecordStore("phoneBook", true);
            System.out.println("The record store phoneBook is opened");
            item = new StringItem("The phoneBook record store was opened
:)", "");
        }
        catch(Exception exception)
        {
            exception.printStackTrace();
            item = new StringItem("The phoneBook record store was not
opened :(", "");
        }
        finally
        {
            try
            {
                rs.close();
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }
    }

```

```

        form.append(item);
        display.setCurrent(form);
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        form = null;
        item = null;
        exitCommand = null;
    }
}

```

Each RecordStore object that represents a table contains a header and a group of data block. Each data block holds the data of one record within the record store. Each data block holds a pointer to the next data block (The data blocks are organized as a linked list). The header of the record store holds the pointers to the first data block and the first free data block space as well as the other data: the number of records within the record store, the record store version number, the time stamp and the next record id that shall be given to new added record. The info within the header can be retrieved using the appropriate getXxx... methods. Some of the information about a record store isn't kept in its header. This information can be retrieved using the appropriate getXxx... methods (examples: getSize() returns the amount of space in bytes that the record store occupies and listRecordStores() returns an array with all the record store names owned by the MIDlet suite).

Records Handling

The record store consists of a collection of records, which remains persistent across multiple invocations. Each record within the record store has an integer record id as its primary key and a byte array that holds its data.

Records can be added\deleted\retrieved and modified using methods, which belong to the RecordStore class:

public int addRecord(byte vec[], int offset, int numOfBytes)

This record adding is an atomic operation. The method returns the recordId for the new added record it adds.

public void deleteRecord(int recordId)

public int getRecord(int recordId, byte vec[], int offSet)

This method returns the number of bytes that were copied into vec starting at offSet.

public byte[] getRecord(int recordId)

public void setRecord(int recordId, byte data[], int offSet, int numOfBytes)

The following midlet presents a simple example of adding new records.

//filename:RecordAddingDemo.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
//All rights reserved. No part of the contents of this program may be

//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.rms.*;
```

```
public class RecordAddingDemo extends MIDlet implements CommandListener
{
    private Display display;
    private Form form;
    private Item item1, item2;
    private Command exitCommand;

    public RecordAddingDemo()
    {
        //creating the exitCommand
        exitCommand = new Command("Exit", Command.EXIT, 1);

        //creating the forms
        form = new Form("Records Adding");
        form.addCommand(exitCommand);

        //setting the listener
        form.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);

        RecordStore rs = null;
        try
        {
            rs = RecordStore.openRecordStore("phoneBook", true);
            System.out.println("The record store phoneBook is opened");
            item1 = new StringItem("The phoneBook record store was opened
:)", "");

            String str = new String("ISRAEL WANT PEACE");
            rs.addRecord(str.getBytes(), 0, str.length());
            System.out.println("Record one : " + str + " was successfully
added");

            str = new String("WORLD WANT PEACE");
            rs.addRecord(str.getBytes(), 0, str.length());
            System.out.println("Record two : " + str + " was successfully
added");
        }
        catch (Exception e)
        {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

```

        item2 = new StringItem("The two records were successfully
added", "");
    }
    catch(Exception exception)
    {
        exception.printStackTrace();
        item1 = new StringItem("The phoneBook record store was not
opened :(", "");
    }
    finally
    {
        try
        {
            if(rs!=null)
            {
                rs.closeRecordStore();
            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
    form.append(item1);
    form.append(item2);
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void commandAction(Command command, Displayable displayable)
{
    if(command==exitCommand)
    {
        exit();
    }
}

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

```

```

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        form = null;
        item1 = null;
        item2 = null;
        exitCommand = null;
        display = null;
    }
}

```

In order to delete a record from a record store you need to call the **deleteRecord()** method. When a record is deleted, its id (its record id) is not reused, and if a new record is added to the record store then its id increases by one over that of the previous last added record. The following midlet adds two records and then deletes them. Each time you run the following midlet the ids of the new added records increase by 1 comparing to the previous last added record (the ids of the deleted records aren't reused).

//filename:RecordAddingDeletingDemo.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
 //All rights reserved. No part of the contents of this program may be
 //reproduced or transmitted in any form or by any means without the
 //written permission of the publisher.

```

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.rms.*;

```

```

public class RecordAddingDeletingDemo extends MIDlet implements CommandListener
{
    private Display display;
    private Form form;

```



```

private Item item1, item2, item3, item4;
private Command exitCommand;

public RecordAddingDeletingDemo()
{
    //creating the exitCommand
    exitCommand = new Command("Exit", Command.EXIT, 1);

    //creating the forms
    form = new Form("Records Adding");
    form.addCommand(exitCommand);

    //setting the listener
    form.setCommandListener(this);

    //getting the display object
    display = Display.getDisplay(this);

    RecordStore rs = null;
    try
    {
        rs = RecordStore.openRecordStore("phoneBook", true);
        System.out.println("The record store phoneBook is opened");
        item1 = new StringItem("The phoneBook record store was opened
:)", "");

        String str = new String("ISRAEL WANT PEACE");
        int id1 = rs.addRecord(str.getBytes(), 0, str.length());
        System.out.println("Record one : " + str + " was successfully
added");

        str = new String("WORLD WANT PEACE");
        int id2 = rs.addRecord(str.getBytes(), 0, str.length());
        System.out.println("Record two : " + str + " was successfully
added");

        item2 = new StringItem("The two records were successfully
added", "");

        rs.deleteRecord(id1);
        System.out.println("Record " + id1 + " was deleted");
        item3 = new StringItem("Record " + id1 + " was deleted", "");
        rs.deleteRecord(id2);
        System.out.println("Record " + id2 + " was deleted");
        item4 = new StringItem("Record " + id2 + " was deleted", "");
    }
    catch(Exception exception)
    {
        exception.printStackTrace();
    }
}

```

```

        item1 = new StringItem("The phoneBook record store was not
opened :(", "");
    }
    finally
    {
        try
        {
            if(rs!=null)
            {
                rs.closeRecordStore();
            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
    form.append(item1);
    form.append(item2);
    form.append(item3);
    form.append(item4);

}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void commandAction(Command command, Displayable displayable)
{
    if(command==exitCommand)
    {
        exit();
    }
}

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

public void pauseApp()
{
}

```

```
public void destroyApp(boolean cond)
{
    form = null;
    item1 = null;
    item2 = null;
    exitCommand = null;
    display = null;
}
}
```

Record Events Handling

The record store produces events in three situations:

A record was added

A record was changed

A record was deleted

Handling these three types of events is done using the **RecordListener** interface that has the following three methods:

```
public void recordAdded(RecordStore recordStore, int recordId)
public void recordChanged(RecordStore recordStore, int recordId)
public void recordDeleted(RecordStore recordStore, int recordId)
```

The **RecordStore** has the following two methods for adding or removing a **RecordListener**:

```
public void addRecordListener(RecordListener listener)
```

```
public void removeRecordListener(RecordListener listener)
```

Unlike a Displayable object or a Form object, the RecordStore object can have multiple registered RecordListeners.

The following midlet adds two records and then deletes them. RecordEvent listener is registered with the RecordStore and presents a message each time a record is deleted.

```
//filename:RecordListenerDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import javax.microedition.rms.*;
```

```
public class RecordListenerDemo extends MIDlet implements CommandListener,  
RecordListener  
{
```

```
    private Display display;  
    private Form form;  
    private Command exitCommand;
```

```
    public RecordListenerDemo()  
    {
```

```
        //creating the exitCommand  
        exitCommand = new Command("Exit", Command.EXIT, 1);
```

```
        //creating the forms  
        form = new Form("Records Adding");  
        form.addCommand(exitCommand);
```

```
        //setting the form listener  
        form.setCommandListener(this);
```

```

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void recordAdded(RecordStore rs, int id)
    {
        System.out.println("record " + id + " was added");
        form.append(new StringItem("record " + id + " was added",""));
    }

    public void recordChanged(RecordStore rs, int id)
    {
        System.out.println("record " + id + " was changed");
    }

    public void recordDeleted(RecordStore rs, int id)
    {
        System.out.println("record " + id + " was deleted");
        form.append(new StringItem("record " + id + " was deleted",""));
    }

    public void exit()
    {
        destroyApp(true);
        notifyDestroyed();
    }

    public void commandAction(Command command, Displayable displayable)
    {
        if(command==exitCommand)
        {
            exit();
        }
    }

    public void startApp()
    {
        display.setCurrent(form);
        RecordStore rs = null;
        try
        {
            rs = RecordStore.openRecordStore("phoneBook", true);
            rs.addRecordListener(this);
            String str = new String("ISRAEL WANT PEACE");
            int id1 = rs.addRecord(str.getBytes(),0,str.length());
        }
    }

```

```

        str = new String("WORLD WANT PEACE");
        int id2 = rs.addRecord(str.getBytes(),0,str.length());
        rs.deleteRecord(id1);
        rs.deleteRecord(id2);
    }
    catch(Exception exception)
    {
        exception.printStackTrace();
    }
    finally
    {
        try
        {
            if(rs!=null)
            {
                rs.closeRecordStore();
            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
    display.setCurrent(form);
}

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    form = null;
    exitCommand = null;
    display = null;
}
}

```

The RecordEnumeration Interface

The right way for retrieving the records that were added to a specific record store is using the RecordEnumeration interface. After all, we can't base the records retrieving on the records' ids since these ids are not consecutive the moment a record is deleted. Each MIDP implementation must provide a concrete class that implements this interface. Calling the enumerateRecords() method that belongs to the RecordStore class returns a RecordEnumeration reference (a reference to object that was instantiated from a concrete class that implements the RecordEnumeration interface).

**public RecordEnumeration enumerateRecords(RecordFileter filter,
RecordComparator comparator, boolean keepUpDated)
throws RecordStoreNotOpenException**

The RecordEnumeration object logically maintains a sequence of the records' ids in a given record store. This object enables iterating over all of the data store records (or a subset if a RecordFilter object has been applied). If a RecordComparator object has been applied then the RecordEnumeration will retrieve the records in the order that the RecordComparator determines. If the record store has been changed (a record has been added\changed\deleted) the RecordEnumeration object isn't automatically updated. In order to keep the RecordEnumeration updated all the time, you should set a record store listener that invokes the rebuild() method on the record store each time a record is added\changed\deleted.

public void rebuild()

Calling this method makes the RecordEnumeration object to be rebuilt so it shall be updated with changes in the record store.

An alternative way, might be calling the **keepUpdated()** method on the RecordEnumeration object.

public void keepUpdated(boolean keepaUpdated)

If keepUpdated is true then the enumerator will keep its enumeration current with any change in the records of the record store (modifying, adding or deleting).

However, using this method has severe performance consequences.

This way of action ensures that the RecordEnumeration object is always updated.

The following example presents a simple use of the rebuild() method.

```
//filename:RecordListenerDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import javax.microedition.rms.*;
```

```
public class RecordListenerDemo extends MIDlet implements CommandListener,  
RecordListener
```

```
{  
    private Display display;  
    private Form form;  
    private Command exitCommand;  
    private RecordEnumeration enumeration;  
  
    public RecordListenerDemo()  
    {
```



```

        //creating the exitCommand
        exitCommand = new Command("Exit", Command.EXIT, 1);

        //creating the forms
        form = new Form("Records Adding");
        form.addCommand(exitCommand);

        //setting the form listener
        form.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void recordAdded(RecordStore rs, int id)
    {
        System.out.println(Thread.currentThread());
        System.out.println("record " + id + " was added");
        form.append(new StringItem("record " + id + " was added", ""));
        enumeration.rebuild();
    }

    public void recordChanged(RecordStore rs, int id)
    {
        System.out.println(Thread.currentThread());
        System.out.println("record " + id + " was changed");
        enumeration.rebuild();
    }

    public void recordDeleted(RecordStore rs, int id)
    {
        System.out.println(Thread.currentThread());
        System.out.println("record " + id + " is deleted");
        form.append(new StringItem("record " + id + " was deleted", ""));
        enumeration.rebuild();
    }

    public void exit()
    {
        destroyApp(true);
        notifyDestroyed();
    }

    public void commandAction(Command command, Displayable displayable)
    {
        if(command==exitCommand)

```

```

        {
            exit();
        }
    }

    public void startApp()
    {
        display.setCurrent(form);
        RecordStore rs = null;
        try
        {
            System.out.println(Thread.currentThread());
            rs = RecordStore.openRecordStore("dotoco", true);
            rs.addRecordListener(this);
            enumeration = rs.enumerateRecords(null,null,false);
            //enumeration.keepUpdated(true);
            //sending true as the third argument to the enumerateRecords
            //method has the same impact.
            String str = new String("ISRAEL WANT PEACE");
            int id1 = rs.addRecord(str.getBytes(),0,str.length());
            str = new String("WORLD WANT PEACE");
            int id2 = rs.addRecord(str.getBytes(),0,str.length());
            rs.deleteRecord(id2);
            display.setCurrent(form);
            int recordId = 0;
            String recordData = null;
            while(enumeration.hasNextElement())
            {
                recordId = enumeration.nextRecordId();
                recordData = new String(rs.getRecord(recordId));
                System.out.println("id=" + recordId
                    + " data=" + recordData);
            }
        }
        catch(Exception exception)
        {
            exception.printStackTrace();
        }
        finally
        {
            try
            {
                if(rs!=null)
                {
                    rs.closeRecordStore();
                }
            }
        }
    }

```

```

        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        form = null;
        exitCommand = null;
        display = null;
    }
}

```

The first call to the `nextRecord()` method returns the record data of the first record in the sequence. Each subsequent call to the `nextRecord()` method returns the next consecutive record's data. Each subsequent call to the `nextRecordId()` method returns the next consecutive record's id.

The other methods the `RecordEnumeration` interface has are:

```

public void destroy()
public boolean hasNextElement()
public boolean hasPreviousElement()
public boolean isKeptUpdated()
public void keepUpdated(boolean keepUpdated)
public byte[] nextRecord()
public int nextRecordId()
public int numRecords()
public byte[] previousRecord()
public int previousRecordId()

```

```
public void rebuild()
public void reset()
```

As was mentioned before, Every device manufacture that implements the J2ME(CLDC\MIDP) must provide a concrete class that implements the RecordEnumeration interface. The RecordEnumeration way of action is similar to a double linked list that enables traversing forward and backward. The reset() method reset the record enumeration object. You should reset the enumerator each time you want to traverse the records - from start - again.

Simulating a Multiple Columns Table

Each record has only one data field, which is represented as a byte array. Even though, it is possible packing multiple fields into the one single record. By doing so, we can simulate a multiple columns table.

Suppose you want to store the details of specific student as a newly added record. Given that each student has the following attributes:

id:int

firstName:String

lastName:String

birthday:long

You can declare the following class:

```
public class Student
```

```

{
    private int id;
    private String firstName;
    private String lastName;
    private long birthday;
}

```

The id of each student can be the id of the record that describes the student. Then it is possible declaring a method that returns a specific student data as a byte array. By invoking this method a byte array that holds the student data is returned. When having this byte array, it can be saved to a record store as a new record. At the same time, it is possible declaring (in the Student class) a constructor that receives a byte array and initialize the new object it creates using the data within that array.

```

public class Student
{
    private int id;
    private String firstName;
    private String lastName;
    private long birthday;

    Student(int id, byte vec[])
    {
        this.id = id;
        initialize(vec);
    }

    public void initialize(byte vec[])
    {
        ByteArrayInputStream bais = null;
        DataInputStream dis = null;
        try
        {
            bais = new ByteArrayInputStream(vec);
            dis = new DataInputStream(bais);
            firstName = dis.readUTF();
            lastName = dis.readUTF();
            birthday = dis.readLong();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}

```

```

    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
    finally
    {
        try
        {
            bais.close();
            dis.close();
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
    }
}

```

```

public byte[] toBytes()
{
    byte data[] = null;
    ByteArrayOutputStream baos = null;
    DataOutputStream dos = null;
    try
    {
        baos = new ByteArrayOutputStream();
        dos = new DataOutputStream(baos);
        dos.writeUTF(firsName);
        dos.writeUTF(lastName);
        dos.writeLong(birthday);
        data = baos.toByteArray();
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    finally

```

```

        {
            try
            {
                baos.close();
                dos.close();
            }
            catch(IOException e)
            {
                e.printStackTrace();
            }
        }
        return data;
    }
}

```

The following midlet is a simple midlet that adds several students data to a given record store and then retrieves them back and display their data to the screen.

//filename:StudentDBDemo.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
 //All rights reserved. No part of the contents of this program may be
 //reproduced or transmitted in any form or by any means without the
 //written permission of the publisher.

```

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.rms.*;
import java.util.*;
import java.io.*;

```

```

public class StudentDBDemo extends MIDlet implements CommandListener,
RecordListener

```

```

{
    private Display display;
    private Form form;
    private Command exitCommand;
    private StudentDB sdb;

    public StudentDBDemo()
    {
        //creating the exitCommand
        exitCommand = new Command("Exit", Command.EXIT, 1);

        //creating the forms
        form = new Form("StudentDB Demo");
        form.addCommand(exitCommand);

        //setting the form listener
        form.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);

        try
        {
            //creating a student data base
            sdb = new StudentDB("technion.db");
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}

```



```
public void recordAdded(RecordStore rs, int id)
{
    System.out.println("record " + id + " was added");
}
```

```
public void recordChanged(RecordStore rs, int id)
{
    System.out.println("record " + id + " was changed");
}
```

```
public void recordDeleted(RecordStore rs, int id)
{
    System.out.println("record " + id + " is deleted");
}
```

```
public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}
```

```
public void commandAction(Command command, Displayable displayable)
{
    if(command==exitCommand)
    {
        exit();
    }
}
```

```
public void startApp()
```

```

{
    display.setCurrent(form);
    try
    {
        Calendar cal = Calendar.getInstance();
        cal.set(Calendar.YEAR, 1970);
        cal.set(Calendar.MONTH, Calendar.JANUARY);
        cal.set(Calendar.DATE, 12);
        sdb.addStudent(new
Student("Haim", "Michael", cal.getTime().getTime()));

        cal = Calendar.getInstance();
        cal.set(Calendar.YEAR, 1970);
        cal.set(Calendar.MONTH, Calendar.FEBRUARY);
        cal.set(Calendar.DATE, 3);
        sdb.addStudent(new
Student("Danniel", "Shabtai", cal.getTime().getTime()));

        cal = Calendar.getInstance();
        cal.set(Calendar.YEAR, 1974);
        cal.set(Calendar.MONTH, Calendar.JANUARY);
        cal.set(Calendar.DATE, 15);
        sdb.addStudent(new
Student("Sandra", "Fox", cal.getTime().getTime()));

        Vector vec = sdb.getAllStudents();
        int numOfStudents = vec.size();
        for(int i=0; i<numOfStudents; i++)
        {
            form.append(new
StringItem("id="+((Student)vec.elementAt(i)).getId(),

```

```

        ((Student)vec.elementAt(i)).getName());
    }
}
catch(Exception exception)
{
    exception.printStackTrace();
}
}

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    form = null;
    exitCommand = null;
    display = null;
}
}

class StudentDB
{
    private RecordStore rs;

    StudentDB(String dbName) throws Exception
    {
        rs = RecordStore.openRecordStore(dbName,true);
    }
}

```

```

int addStudent(Student std) throws Exception
{
    byte []temp = std.toBytes();
    return rs.addRecord(temp,0,temp.length);
}

```

```

Vector getAllStudents()
{
    RecordEnumeration enum = null;
    Vector vec = new Vector();
    try
    {
        enum = rs.enumerateRecords(null, null, false);
        int tempId = 0;
        while(enum.hasNextElement())
        {
            tempId = enum.nextRecordId();
            byte data[] = rs.getRecord(tempId);
            vec.addElement(new Student(tempId,data));
        }
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    finally
    {
        if(enum!=null)
        {
            enum.destroy();
        }
    }
}

```

```
        }  
    }  
    return vec;  
}  
}
```

```
class Student  
{  
    private int id;  
    private String firstName;  
    private String lastName;  
    private long birthday;  
  
    Student(String fName, String lName, long birth)  
    {  
        firstName = fName;  
        lastName = lName;  
        birthday = birth;  
    }  
  
    Student(int id, byte vec[])  
    {  
        this.id = id;  
        initialize(vec);  
    }  
  
    String getName()  
    {  
        return firstName + lastName;  
    }  
}
```

```
int getId()
{
    return id;
}

void initialize(byte vec[])
{
    ByteArrayInputStream bais = null;
    DataInputStream dis = null;
    try
    {
        bais = new ByteArrayInputStream(vec);
        dis = new DataInputStream(bais);
        firstName = dis.readUTF();
        lastName = dis.readUTF();
        birthday = dis.readLong();
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
    finally
    {
        try
        {
            bais.close();
            dis.close();
        }
        catch(IOException e)
        {

```

```

        e.printStackTrace();
    }
}

byte[] toBytes()
{
    byte data[] = null;
    ByteArrayOutputStream baos = null;
    DataOutputStream dos = null;
    try
    {
        baos = new ByteArrayOutputStream();
        dos = new DataOutputStream(baos);
        dos.writeUTF(firstName);
        dos.writeUTF(lastName);
        dos.writeLong(birthday);
        data = baos.toByteArray();
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    finally
    {
        try
        {
            baos.close();
            dos.close();
        }
    }
    catch(IOException e)

```

```
        {  
            e.printStackTrace();  
        }  
    }  
    return data;  
}  
}
```

The RecordComparator and the RecordFilter

When retrieving the records from a given record store using a RecordStoreEnumeration object it is possible to set a RecordFilter to filter the retrieved records. Another option is setting a RecordComparator object in order the records sorted according to the RecordComparator rule.

The RecordFilter interface has one method:

public boolean matches(byte []vec)

This method, when invoked receives a specific record data and returns true/false according to the question whether the record is included within the returned enumeration or not. You can declare a class that implements this interface and use its instantiated object to filter the returned records.

The RecordComparator interface has one method:

public int compare(byte []vec1, byte []vec2)

This method returns one of the following integer values:

`RecordComperator.PRECEDES`, `RecordComperator.FOLLOWS` or

`RecordComperatoe.EQUIVALENT`. You can declare a class that implements this interface and instantiate it in order to sort the returned records.

The following midlet presents a simple use of these two interfaces. The following midlet adds several students to the record store and retrieve them back sorted and without including the details of the students that their first name starts with 'A'.

```
//filename:StudentsDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
```

```
//All rights reserved. No part of the contents of this program may be
```

```
//reproduced or transmitted in any form or by any means without the
```

```
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;
```

```
import javax.microedition.lcdui.*;
```

```
import javax.microedition.rms.*;
```

```
import java.util.*;
```

```
import java.io.*;
```

```
public class StudentsDemo extends MIDlet implements CommandListener,  
RecordListener
```

```
{
```

```
    private Display display;
```

```

private Form form;
private Command exitCommand;
private SudentDBManager sdb;

public StudentsDemo()
{
    //creating the exitCommand
    exitCommand = new Command("Exit", Command.EXIT, 1);

    //creating the forms
    form = new Form("SudentDBManager Demo");
    form.addCommand(exitCommand);

    //setting the form listener
    form.setCommandListener(this);

    //getting the display object
    display = Display.getDisplay(this);

    try
    {
        //creating a student data base
        sdb = new SudentDBManager("kedem.db");
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}

public void recordAdded(RecordStore rs, int id)

```

```
{
    System.out.println("record " + id + " was added");
}

public void recordChanged(RecordStore rs, int id)
{
    System.out.println("record " + id + " was changed");
}

public void recordDeleted(RecordStore rs, int id)
{
    System.out.println("record " + id + " is deleted");
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void commandAction(Command command, Displayable displayable)
{
    if(command==exitCommand)
    {
        exit();
    }
}

public void startApp()
{
    display.setCurrent(form);
}
```

```

try
{
    sdb.addSudent(new Sudent("Haim","Michael"));
    sdb.addSudent(new Sudent("Danniel","Shabtai"));
    sdb.addSudent(new Sudent("Sandra","Fox"));
    sdb.addSudent(new Sudent("Avi","Cohen"));
    sdb.addSudent(new Sudent("David","Fredy"));
    sdb.addSudent(new Sudent("Amnon","Shomron"));
    sdb.addSudent(new Sudent("Ziv","Gilboa"));
    sdb.addSudent(new Sudent("Rozen","Tishbi"));
    sdb.addSudent(new Sudent("Barbara","Zamir"));

    Vector vec = sdb.getAllSudents();
    int numOfSudents = vec.size();
    for(int i=0; i<numOfSudents; i++)
    {
        form.append(new
StringItem("id="+((Sudent)vec.elementAt(i)).getId(),

        ((Sudent)vec.elementAt(i)).getName()));
    }
}
catch(Exception exception)
{
    exception.printStackTrace();
}

}

public void pauseApp()
{
}

```

```

public void destroyApp(boolean cond)
{
    form = null;
    exitCommand = null;
    display = null;
}
}

```

```

class SudentsFilter implements RecordFilter
{
    public boolean matches(byte []data)
    {
        Sudent std = new Sudent(data);
        return !std.getName().startsWith("A");
    }
}

```

```

class SudentsSorter implements RecordComparator
{
    public int compare(byte vec1[], byte vec2[])
    {
        Sudent std1, std2;
        int result = RecordComparator.EQUIVALENT;
        std1 = new Sudent(vec1);
        std2 = new Sudent(vec2);
        int num = std1.getName().compareTo(std2.getName());
        if(num<0)
        {

```

```

        result = RecordComparator.PRECEDES;
    }
    else if(num>0)
    {
        result = RecordComparator.FOLLOWS;
    }
    else if(num==0)
    {
        result = RecordComparator.EQUIVALENT;
    }
    return result;
}
}

```

class SudentDBManager

```

{
    private RecordStore rs;

    SudentDBManager(String dbName) throws Exception
    {
        rs = RecordStore.openRecordStore(dbName,true);
    }

    int addSudent(Sudent std) throws Exception
    {
        byte []temp = std.toBytes();
        return rs.addRecord(temp,0,temp.length);
    }

    Vector getAllSudents()

```

```

{
    RecordEnumeration enum = null;
    Vector vec = new Vector();
    try
    {
        enum = rs.enumerateRecords(
            new SudentsFilter(),
            new SudentsSorter(),
            false);

        int tempId = 0;
        while(enum.hasNextElement())
        {
            tempId = enum.nextRecordId();
            byte data[] = rs.getRecord(tempId);
            vec.addElement(new Sudent(tempId,data));
        }
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    finally
    {
        if(enum!=null)
        {
            enum.destroy();
        }
    }
    return vec;
}
}

```

```
class Student
{
    private int id;
    private String firstName;
    private String lastName;

    Student(String fName, String lName)
    {
        firstName = fName;
        lastName = lName;
    }

    Student(int id, byte vec[])
    {
        this.id = id;
        initialize(vec);
    }

    Student(byte vec[])
    {
        initialize(vec);
    }

    String getName()
    {
        return firstName + " " + lastName;
    }

    int getId()
    {
```



```

        return id;
    }

    void initialize(byte vec[])
    {
        ByteArrayInputStream bais = null;
        DataInputStream dis = null;
        try
        {
            bais = new ByteArrayInputStream(vec);
            dis = new DataInputStream(bais);
            firstName = dis.readUTF();
            lastName = dis.readUTF();
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
        finally
        {
            try
            {
                bais.close();
                dis.close();
            }
            catch(IOException e)
            {
                e.printStackTrace();
            }
        }
    }
}

```

```
byte[] toBytes()
{
    byte data[] = null;
    ByteArrayOutputStream baos = null;
    DataOutputStream dos = null;
    try
    {
        baos = new ByteArrayOutputStream();
        dos = new DataOutputStream(baos);
        dos.writeUTF(firstName);
        dos.writeUTF(lastName);
        data = baos.toByteArray();
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    finally
    {
        try
        {
            baos.close();
            dos.close();
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
    }
    return data;
}
```

}

}

Chapter 5:

Networking

- • [Introduction](#)
- • [Networking using Sockets](#)
- • [Networking using Datagram](#)
- • [Networking using HttpConnection](#)

Introduction

Big differences exist in network programming between J2ME and J2SE. While the J2SE runs on computers with faster CPU and larger memory, the J2ME runs on smaller devices with slower CPU and less memory. Furthermore, the J2ME needs to support a variety of devices with different networking and file I/P capabilities. The Generic Connection Framework (part of the J2ME(CLDC\MIDP)) meets the challenges of different devices with different capabilities by generalizing the functionality of J2SE's network and file I\O classes into several interfaces. Each MIDP implementation implements the relevant interfaces (based on the device capabilities). These classes and interfaces are included in a single package: `javax.microedition.io`.

Notice that when you run an example from this chapter, the device (or the device emulator) on which you run it needs to support the protocol the example uses.

The Generic Connection Framework supports the following ways of communication:

HTTP

Sockets

Datagrams (UDP)

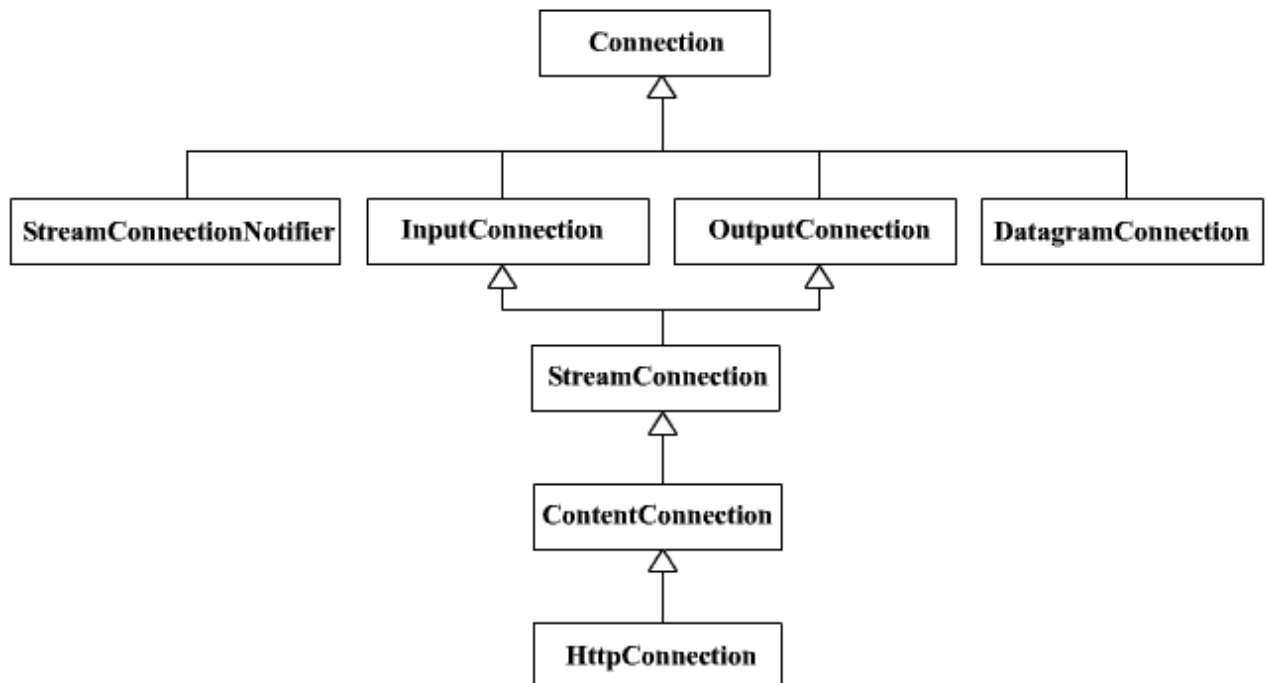
Serial Port

File

Each connection (no matter what the communication form is) is created by calling the `Connector.open()` static method:

```
Connector.open("http://www.zindell.com");  
Connector.open("socket://localhost:8080");  
Connector.open("datagram://www.webnow.com");  
Connector.open("comm.:0;baudrate=9600");  
Connector.open("file:/meText.dat");
```

The following diagram presents the interfaces hierarchy that exists in the *Generic Connection Framework*:



Since the HTTP communication is expected to be available in all of the MIDP implementations (all the devices), the **HttpConnection** interface must always be implemented. This means, that every MIDP implementation will have a concrete class that implements the **HttpConnection** interface.

The Connector class is the one and only concrete class that the *Generic Connection Framework* specifies. Every network connection is created by calling the `open()` static method which belongs to that class.

The `open()` method receives a string that indicate the connection form we want to create. This string has the following format:

protocol : target params

The protocol value indicates the connection form:

http

socket

datagram

comm.

file

Syntax examples for using the `open` method in order to create each of these communication forms were given in the beginning of this chapter.

The target can be each of the following: domain name, network port number, file name etc...

The params are additional information needed to complete the connection. They are optional.

The Connector class contains the following seven static methods:

public static Connection open(String str)

public static Connection open(Strgin str, int mode)

public static Connection open(String str, int mode, boolean timeout)

public static DataInputStream open(String str)

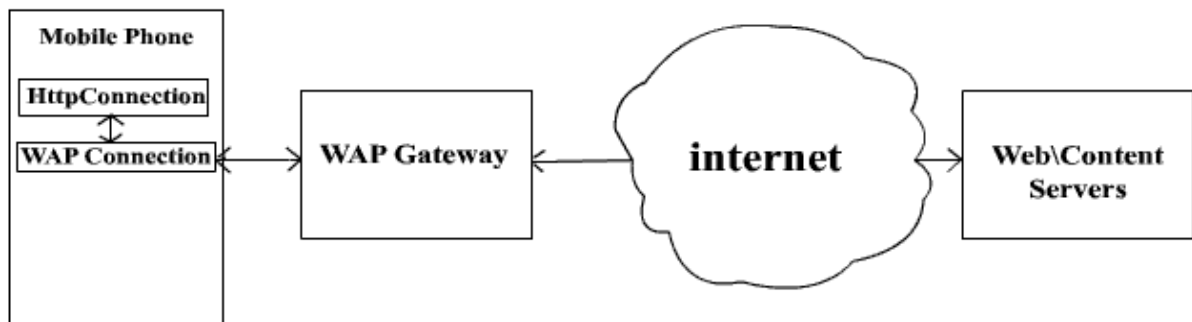
public static DataOutputStream open(String str)

public static InputStream openInputStream(String str)

public static OutputStream openOutputStream(Strgin str)

Three different integer finals were declared in the Connector class for specifying the connection type: READ, READ_WRITE, WRITE. If the mode isn't mentioned then the default takes place: Connector.READ_WRITE.

Since the HttpURLConnection interface must be implemented in every MIDP implementation, a concrete class that implements HttpURLConnection must be declared. If the device doesn't support HTTP then the underlying support mechanism for HTTP connections might be stack on top of another non-IP based protocols. Further more, there will be a need in a one more gateway server that convert the cellular telephones protocol (WAP for instance) to TCP\IP protocol (and vice-versa) to enable surfing the net. The following figure demonstrates this case.



Networking using Sockets

When two computers connect with other using sockets, each one of them has a socket, one end point of a two-way communication. Using the sockets, among other operations, the device can connect a SMTP and POP3 email servers. However, sockets support doesn't exist in every device.

When using sockets, the two computers communicate through the sockets each one of them has. Before the two sockets are created, one of the two computers (the server) listens and waits for a connection request to arrive. Then, the other computer (the client) shall try to create the connection by sending to the first computer a request for establishing the connection. Once the two sockets are created they can be used for sending\receiving data. The J2ME(CLDC\MIDP) has three classes that describe different connection types: `InputConnection`, `OutputConnection` and `StreamConnection`. The `InputConnection` is used for receiving data, The `OutputConnection` is used for sending data. The `StreamConnection` is used both for sending and for receiving data.

The steps of creating a connection with a remote server are as follows:

Step 1

The socket connection is opened with the remote computer(server or other mobile device) by calling the `open()` static method that was declared in the `Connector` class. The string that is sent as argument to the open method should starts with "socket" to indicate that the needed connection is with sockets and its type is one of the following: `InputConnection`, `OutputConnection` or `StreamConnection`.

Casting to one of these three types should be done to the returned reference that the open() method returns.

Step 2

Now, on the Connection object that was created

(InputConnection\OutputConnection\StreamConnection) one of the following methods: openDataOutputStream, openDateInputStream, openOutputStream, openInputStream) will be invoked. Each of these methods return an stream reference.

Step 3

Data can be sent and received to\from the remote computer via the socket connections by calling read and writes methods on the input\output streams.

Step 4

When sending\receiving the data is finished the streams should be closed.

The following midlet presents a simple network communication with a remote device on which a little server sends back the US\$/NIS exchange rate. The midlet is the client and your computer (localhost) is the server.

```
//filename:NetworkingDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;  
import javax.microedition.io.*;
```

```
public class NetworkingDemo extends MIDlet  
{
```

```

private Display display;
private Form form;
private Command exitCommand, getCommand;
private String connectionDetails = "socket://localhost:1300";
private InputConnection ic;
private DataInputStream dis;

public NetworkingDemo()
{
    //create the backCommand, okCommand and the exitCommand
    exitCommand = new Command("Exit", Command.EXIT, 1);
    getCommand = new Command("Get", Command.SCREEN, 1);
    System.out.println("exit command and get command were created");

    //creating the form
    form = new Form("EXCHANGE RATE");
    form.addCommand(exitCommand);
    form.addCommand(getCommand);
    System.out.println("exit command and get command were appended to the
form");

    //instantiating the listener
    CommandListener listener = new CommandListener()
    {
        public void commandAction(Command c,
            Displayable d)
        {
            if(c==exitCommand)
            {
                System.out.println("exitCommand was
pressed");

                exit();
            }
            else
            {
                if(c==getCommand)
                {
                    System.out.println("getCommand
was pressed");

                    try
                    {
                        //establishing a connection
                        System.out.println("about to
get a stream connection");

```

```

        ic =
        (InputConnection)Connector.open(connectionDetails,Connector.READ);
        System.out.println("input
connection was created");

        //input stream is created on
        //of the stream connection
        dis =
        System.out.println("input
stream was created");

        //receiving data
        StringBuffer sb = new
        StringBuffer();

        int input = dis.read();
        while(input!=-1)
        {

            sb.append((char)input);

            input = dis.read();
        }

        System.out.println("dataReceived:"+sb.toString());

        //displaying the received data
        if(form.size()!=0)
        {
            form.delete(0);
        }
        form.append(new
        StringItem("DOLLAR NIS exchange rate is ", sb.toString()));
    }
    catch(Exception e)
    {
        if(form.size()!=0)
        {
            form.delete(0);
        }
        form.append(new
        StringItem("problem:",e.getMessage()));
    }
    finally

```

were allocated

```
        {
            //releasing resources that
            try
            {
                if(dis!=null)
                {
                    dis.close();
                }
            }
            catch(IOException e)
            {
                e.printStackTrace();
            }
            try
            {
                if(ic!=null)
                {
                    ic.close();
                }
            }
            catch(IOException e)
            {
                e.printStackTrace();
            }
        }
    }
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

}

//setting the listeners
form.setCommandListener(listener);

//getting the display object
display = Display.getDisplay(this);
}

public void startApp()
```

```

        {
            display.setCurrent(form);
            System.out.println("form was set as the current screen");
        }

        public void pauseApp()
        {
        }

        public void destroyApp(boolean cond)
        {
            display = null;
            form = null;
            exitCommand = null;
        }
    }
}

```

//filename:TCPSimpleServer.java

//Copyright (c) 2000 Haim Michael & Zindell Publishing House, Ltd.
 //All rights reserved. No part of the contents of this program may be
 //reproduced or transmitted in any form or by any means without the
 //written permission of the publisher.

```

import java.io.*;
import java.net.*;
import java.util.*;

public class TCPSimpleServer
{
    private ServerSocket sSoc;
    public static final int PORT = 1300;

    public static void main(String args[]) throws IOException
    {
        TCPSimpleServer server = new TCPSimpleServer();
        try
        {
            server.go();
        }
        catch(IOException e)
        {
        }
    }
}

```

```

        {
            System.out.println("The Server start failed");
            e.printStackTrace();
        }
    }

    public void go() throws IOException
    {
        Socket soc = null;
        sSoc = new ServerSocket(PORT);
        OutputStream os = null;
        DataOutputStream dos = null;

        while(true)
        {
            try
            {

                //creating a socket
                soc = sSoc.accept();
                System.out.println("socket was created");

                //creating an output stream
                os = soc.getOutputStream();
                System.out.println("output stream was created");

                //creating a dataoutput stream
                String out = " 4.22400";
                dos = new DataOutputStream(os);
                os.write(out.getBytes());
                os.flush();
                System.out.println("out:"+out);
            }
            catch(IOException e)
            {
                e.printStackTrace();
            }
            finally
            {
                try
                {
                    os.close();
                }
                catch(IOException e)
                {
                    e.printStackTrace();
                }
            }
        }
    }

```

```

    }
    try
    {
        dos.close();
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
    try
    {
        soc.close();
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
}
}
}
}

```

Networking using Datagrams(UDP)

Sending data using UDP (User Datagrams Protocol) means the data is sent over the network as a packet that its arrival, arrival time as well as its content integrity are not guaranteed. Unlike using sockets communication, when using the UDP is connectionless (no dedicated open connection exists between the client and the server).

When a device wants to send data to another device (using UDP) it first has to build a datagram packet with the destination information (Internet address and a

port number) and the data the device want to send. Unlike the socket communication, when using the UDP, the network implementation doesn't perform any error checking, acknowledgement of the packets arrival or sequencing the packets you send so their order will be guaranteed in their arrival. Because the UDP isn't a guaranteed delivery protocol it is not suitable for every application.

Using UDP is suitable in the following cases:

When the speed is more critical than transmitting every bit correctly (real time audio\video for instance).

When the information is sent frequently (server that sends the temperature every 30 seconds).

Sometimes, the given wireless device doesn't support socket communication. In these cases there is no other better alternative.

Two J2ME(CLDC\MIDP) provides two interfaces that support the UDP communication: `DatagramConnection` and `Datagram`.

The **`DatagramConnection`** is one of the interfaces that extends the `Connection` interfaces. This interface describes a connection based on the UDP protocol. The `DatagramConnection` extends from `Connection` just the `close()` method:

```
public void close() throws IOException
```

The `DatagramConnection` adds the following methods:

```
public int getMaximumLength()
```

```
public int getMinimumLength()
```

```
public Datagram newDatagram(byte vec[], int size)
```

```
public Datagram newDatagram(byte vec[], int size, String addr)
```

```
public Datagram newDatagram(int size)
```

```
public Datagram newDatagram(int size, String addr)
```

```
public void receive(Datagram datagram)
```

```
public void send(Datagram datagram)
```

The **Datagram** interface describes a packet of data that can be used either as a placeholder of a message received from another computer or as a placeholder for message you want to send. The Datagram interface extends the DataInput and the DataOutput interfaces that provide the methods for the necessary read and write operations to the data that the datagram object holds.

The Datagram interface extends the following methods from the DataInput and the DataOutput interfaces:

```
public void write(byte []vec)
```

```
public void write(byte []vec, int off, int length)
```

```
public void write(int value)
```

```
public void writeBoolean(boolean value)
```

```
public void writeByte(int value)
```

```
public void writeChar(int value)
```

```
public void writeChars(String str)
```

```
public void writeInt(int value)
```

```
public void writeLong(long value)
```

```
public void writeShort(int value)
```

```
public void writeUTF(String str)
```

```
public boolean readBoolean()
```

```
public byte readByte()
```

```
public char readChar()
```

```
public void readFully(byte vec[])
```

```
public void readFully(byte vec[], int off, int length)
```

```
public int readInt()
```

```
public long readLong()
```

```
public short readShort()
public int readUnsignedByte()
public int readUnsignedShort()
public String readUTF()
public int skipBytes(int number)
public boolean readBoolean()
public byte readByte()
public char readChar()
public void readFully(byte vec[])
public void readFully(byte vec[])
public int readInt()
public long readLong()
public short readShort()
public int readUnsignedByte()
public int readUnsignedShort()
public String readUTF()
public void skipBytes(int num)
```

The Datagram interface also defines the following UDP specific methods:

```
public String getAddress()
public byte[] getData()
public int getLength()
public int getOffset()
public void reset()
public void setAddress(Datagram ref)
public void setAddress(String addr)
public void setData(byte[] buffer, int offset, int length)
public void setLength(int length)
```

The first step in datagram communication is creating the datagram connection.

Like the other types of connections, calling the static open method in Connector creates the datagram connection. The datagram connection can be created in one of the two modes: server mode or client mode.

To get a datagram connection in a **client mode** you should send to the open method a string that includes the host name.

Example:

```
DatagramConnection dc =  
(DatagramConnection)Connector.open("datagram://122.32.5.222:1300");
```

To get a datagram connection in a **server mode** you should send to the open method a string that doesn't include the host name.

Example:

```
DatagramConnection dc =  
(DatagramConnection)Connector.open("datagram://:1300");
```

When having a server datagram connection, information can be both sent and received. When having a client datagram connection, information can only be sent. When using a client datagram connection the datagram of information to be sent must include the destination hostname and the destination port number. The port number in "server mode" (the host name is not specified) is the receiving port. The port number in "client mode" (the host name is specified) is the target port. When the connection is in server mode then the same port is used for both receiving and sending. When the connection is in client mode the port the reply is sent back to is allocated dynamically.

When the datagram connection is created you should construct a datagram with a message body, destination address and the port and send it. The following commands will create a datagram to send:

```
byte msg[] = "Hello".getBytes();  
Datagram sendDg = dc.newDatagram(msg, msg.length,  
"datagram://123.23.22.1:1300");  
dc.send(sendDg);
```

Now, it is possible constructing a received datagram with a pre allocated array in which the received message will be saved. The following code presents a simple creation of a datagram in which the received message will be saved.

```
Datagram receiveDatagram = dc.newDatagram(dc.getMaximumLength());  
dc.receive(receiveDatagram);
```

The following example presents a midlet that works as a client and a stand-alone application that functions as a server. The midlte sends to the server the user name and receives back from the server a greeting message. Both the midlet and the server run on the same machine. The first code is the UDP stand-alone application. The second code is the UDP midlet client.

```
//filename:UDPServer.java
```

```
//Copyright (c) 2000 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import java.io.*;  
import java.net.*;  
import java.util.*;
```

```

public class UDPServer
{
    public byte[] getMsg(String name)
    {
        return ("HELLO " + name).getBytes();
    }

    public void go()
    {
        DatagramSocket datagramSocket;
        DatagramPacket inputDatagramPacket;
        DatagramPacket outputDatagramPacket;
        InetAddress clientAdd;
        int clientPort;
        byte[] incomingMsg = new byte[10];
        byte[] outgoingMsg;

        System.out.println("The server is ON");

        while(true)
        {
            try
            {
                System.out.println("The server is ON");
                datagramSocket = new DatagramSocket(1400);
                System.out.println("Server is listening to port 1400");

                inputDatagramPacket = new DatagramPacket(
incomingMsg,
incomingMsg.length);
                datagramSocket.receive(inputDatagramPacket);

                System.out.println("a datagram packet was received");

                String rcvMsg = new String(
inputDatagramPacket.getData(),
                                0,
inputDatagramPacket.getLength());

                System.out.println("The server has received : "
                                + rcvMsg);

                clientAdd = inputDatagramPacket.getAddress();

```

```

        clientPort = inputDatagramPacket.getPort();

        byte result[] = getMsg(rcvMsg);
        outputDatagramPacket = new DatagramPacket(    result,

result.length,

clientAdd,

clientPort);

        datagramSocket.send(outputDatagramPacket);
        System.out.println("The server sent back " + new
String(result));

        datagramSocket.close();

    }
    catch(IOException exception)
    {
        exception.printStackTrace();
    }
}

public static void main(String args[])
{
    UDPServer udpServer = new UDPServer();
    udpServer.go();
}
}

```

//filename:UDPClientDemo.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.

```

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

```

```

import java.io.*;
import javax.microedition.io.*;

public class UDPClientDemo extends MIDlet
{
    private Display display;
    private Form form;
    private Command exitCommand, sendCommand;
    private DatagramConnection dc;
    private Datagram sendDatagram;
    private Datagram receiveDatagram;

    public UDPClientDemo()
    {
        //create the backCommand, okCommand and the exitCommand
        exitCommand = new Command("Exit", Command.EXIT, 1);
        sendCommand = new Command("Send", Command.SCREEN, 1);
        System.out.println("exit command and get command were created");

        //creating the form
        form = new Form("UDP-NET");
        form.addCommand(exitCommand);
        form.addCommand(sendCommand);
        System.out.println("exit command and get command were appended to the
form");

        //instantiating the listener
        CommandListener listener = new CommandListener()
        {
            public void commandAction(Command c, Displayable d)
            {
                if(c==exitCommand)
                {
                    System.out.println("exitCommand was
pressed");
                    exit();
                }
                else
                {
                    if(c==sendCommand)
                    {
                        System.out.println("sendCommand
was pressed");
                        String sendMsg = "HAIM";

```



```

StringItem("Send:\n",sendMsg));

datagram connection

(DatagramConnection)Connector.open("datagram://:1",Connector.READ_WRITE);

and send it

dc.newDatagram(sendMsg.getBytes(), sendMsg.length(), "datagram://localhost:1400");
dc.send(sendDatagram);
System.out.println(sendMsg
+ " was sent");

receiveDatagram and fill it with the received message

dc.newDatagram(20);

datagram was created");

datagram was received");

String(receiveDatagram.getData());

System.out.println("received:"+received);

StringItem("Receive:\n", received));

StringItem("problem:",e.getMessage()));

were allocated

```

```

form.append(new

try
{
    //creating a client type

    dc =

    //creating the sendDatagram

    sendDatagram =
    dc.send(sendDatagram);
    System.out.println(sendMsg

    //creating the

    receiveDatagram =

    System.out.println("a receive

    dc.receive(receiveDatagram);
    System.out.println("a

    //receiving data
    String received = new

    //displaying the received data
    form.append(new

}
catch(Exception e)
{
    form.append(new

}
finally
{
    //releasing resources that

```

```

        try
        {
            if(dc!=null)
            {
                dc.close();
            }
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
    }

}

}

}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

}

;

//setting the listeners
form.setCommandListener(listener);

//getting the display object
display = Display.getDisplay(this);
}

public void startApp()
{
    display.setCurrent(form);
    System.out.println("form was set as the current screen");
}

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    display = null;
    form = null;
}

```

```
        exitCommand = null;  
        sendCommand = null;  
    }  
}
```

Networking using HttpURLConnection

Using the `HttpURLConnection` has several advantages. The main advantage is that all of the MID devices support using the `HttpURLConnection` (Unlike the other forms of networking). Other advantages include the following:

The HTTP communication can easily go through firewalls.

Using the HTTP protocol and Java Servlets\JSP connecting the midlet with an application server can be very easy.

Using the HTTP and XML it delivering different types of information to the midlet can be very easy.

The HTTP is a protocol in which there is a respond to each request that is sent. Parameters that describe the request must be set in the `HttpURLConnection` object before the request is sent. The `HttpURLConnection` can be in each one of the following three states:

Setup - The connection with the server has not been made yet.

Connected - The connection has been made, the parameters that describe the request were set in the `HttpURLConnection` object and the request was sent. This state last until the response arrival.

Closed - The connection has been closed.

The `URLConnection` defines different methods. These methods can be grouped into three groups according to the state in which the `URLConnection` can be when they are called.

The following methods can be called when the `URLConnection` object is in its set up state:

```
public void setRequestMethod(String method)
public void setRequestProperty(String key, String value)
```

The transition from setup to connected is caused when a method (one of the following is called):

```
public long getDate()
public long getExpiration()
public String getHeaderField()
public String getHeaderField(int n)
public String getHeaderFieldKey(int n)
public long getLastModified()
public long getHeaderFieldDate(String str, long num)
public long getHeaderFieldInt(String str, int def)
public String getResponseMessage()
public String getResponseCode()
public String getEncoding()
public String getType()
public long getLength()
public InputStream openInputStream()
public OutputStream openOutputStream()
public DataInputStream openDataInputStream()
```

```
public DataOutputStream openDataOutputStream()
```

The following methods might be invoked while the connection is open (Connected state):

```
public void close()
```

```
public String getRequestMethod()
```

```
public String getRequestProperty(String key)
```

```
public String getURL()
```

```
public String getProtocol()
```

```
public String getHost()
```

```
public String getFile()
```

```
public String getRef()
```

```
public String getPort()
```

```
public String getQuery()
```

It is important to notice the interfaces that the `HttpConnection` extends: `Connection`, `InputConnection`, `OutputConnection`, `StreamConnection` and `ContentConnection`. Sometimes, the functionality we need already resides in the methods that the super interfaces declare.

The following midlet presents a simple data retrieving from a certain web page. In order to check this midlet with a real web page just replace the `urlStr` value we set with another one. The following midlet assumes that a web server is running on `localhost` listening to port number 8080 and holding the `TA.html` file.

```
//filename:SimpleHttpDemo1.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be
```

//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;
```

```
public class SimpleHttpDemo1 extends MIDlet
{
    private Display display;
    private Form form;
    private Command exitCommand, retrieveCommand;

    public SimpleHttpDemo1()
    {
        //create the backCommand, okCommand and the exitCommand
        exitCommand = new Command("Exit", Command.EXIT, 1);
        retrieveCommand = new Command("Retrieve", Command.SCREEN, 1);

        //creating the form
        form = new Form("HTTPDEMO1");
        form.addCommand(exitCommand);
        form.addCommand(retrieveCommand);

        //instantiating the listener
        CommandListener listener = new CommandListener()
        {
            public void commandAction(Command c, Displayable d)
            {
                if(c==exitCommand)
                {
                    System.out.println("exitCommand was
pressed");
                    exit();
                }
                else
                {
                    if(c==retrieveCommand)
                    {
                        System.out.println("retrieveCommand was pressed");
                        StreamConnection sc = null;
                        InputStream is = null;
```

```

try
{
    StringBuffer sb = new
StringBuffer();
    sc =
(StreamConnection)Connector.open("http://localhost:8080/TA.html");
    is = sc.openInputStream();
    int ch = is.read();
    while(ch!=-1)
    {
        sb.append((char)ch);
        ch = is.read();
    }
    form.append(new
StringItem("TA.html content is : ", sb.toString()));
}
catch(Exception exception)
{
    form.append(new
StringItem("exception:",exception.getMessage()));
}
finally
{
    try
    {
        if(sc!=null)
        {
            sc.close();
        }
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    try
    {
        if(is!=null)
        {
            is.close();
        }
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
}

```

```

        }
    }
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

}

;

//setting the listeners
form.setCommandListener(listener);

//getting the display object
display = Display.getDisplay(this);
}

public void startApp()
{
    display.setCurrent(form);
    System.out.println("form was set as the current screen");
}

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    display = null;
    form = null;
    exitCommand = null;
    retrieveCommand = null;
}
}

```

The following example does the same using the *ContentConnction* object. The *ContentConnection* interface extends the *StreamConnection* and adds several methods that help in retrieving data using the HTTP protocol.


```
//filename:SimpleHttpDemo1.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;  
import javax.microedition.io.*;
```

```
public class SimpleHttpDemo1 extends MIDlet  
{  
    private Display display;  
    private Form form;  
    private Command exitCommand, retrieveCommand;  
  
    public SimpleHttpDemo1()  
    {  
        //create the backCommand, okCommand and the exitCommand  
        exitCommand = new Command("Exit", Command.EXIT, 1);  
        retrieveCommand = new Command("Retrieve", Command.SCREEN, 1);  
  
        //creating the form  
        form = new Form("HTTPDEMO1");  
        form.addCommand(exitCommand);  
        form.addCommand(retrieveCommand);  
  
        //instantiating the listener  
        CommandListener listener = new CommandListener()  
        {  
            public void commandAction(Command c, Displayable d)  
            {  
                if(c==exitCommand)  
                {  
                    System.out.println("exitCommand was  
pressed");  
                    exit();  
                }  
                else  
                {  
                    if(c==retrieveCommand)
```

```

        {
            System.out.println("retrieveCommand was pressed");
            ContentConnection cc = null;
            InputStream is = null;
            try
            {
                String result = null;
                cc =
(ContentConnection)Connector.open("http://localhost:8080/TA.html");
                int length =
(int)cc.getLength();

                if(length>0)
                {
                    is =
cc.openInputStream();

                    byte data[] = new
byte[length];

                    int wereRead =
is.read(data);

                    if(wereRead!=length)
                    {
                        throw new
IOException("length unknown");

                    }
                    result = new
String(data);

                }
                else
                {
                    StringBuffer sb = new
StringBuffer();

                    int temp = is.read();
                    while(temp!=-1)
                    {

                        sb.append((char)temp);

                        temp =
is.read();

                    }
                    result = sb.toString();

                }
                form.append(new
StringItem("TA.html content is : ", result));

            }
            catch(Exception exception)

```

```

        {
            form.append(new
StringItem("exception:",exception.getMessage()));
        }
        finally
        {
            try
            {
                if(cc!=null)
                {
                    cc.close();
                }
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
            try
            {
                if(is!=null)
                {
                    is.close();
                }
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }
    }
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

;

//setting the listeners
form.setCommandListener(listener);

//getting the display object

```

```

        display = Display.getDisplay(this);
    }

    public void startApp()
    {
        display.setCurrent(form);
        System.out.println("form was set as the current screen");
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        exitCommand = null;
        retrieveCommand = null;
    }
}

```

The following example uses the `HttpConnection` object in order to retrieve information on a specific page (`TA.html`, the page that was retrieved in the last two examples).

```
//filename:SimpleHttpDemo1.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.
```

```

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;

public class SimpleHttpDemo1 extends MIDlet
{

```

```

private Display display;
private Form form;
private Command exitCommand, retrieveCommand;

public SimpleHttpDemo1()
{
    //create the backCommand, okCommand and the exitCommand
    exitCommand = new Command("Exit", Command.EXIT, 1);
    retrieveCommand = new Command("Retrieve", Command.SCREEN, 1);

    //creating the form
    form = new Form("HTTPDEMO1");
    form.addCommand(exitCommand);
    form.addCommand(retrieveCommand);

    //instantiating the listener
    CommandListener listener = new CommandListener()
    {
        public void commandAction(Command c, Displayable d)
        {
            if(c==exitCommand)
            {
                System.out.println("exitCommand was
pressed");
                exit();
            }
            else
            {
                if(c==retrieveCommand)
                {
                    System.out.println("retrieveCommand was pressed");
                    int port = 0;
                    String host = null;
                    String protocol = null;
                    HttpConnection hc = null;
                    InputStream is = null;
                    try
                    {
                        String result = null;
                        hc =
(HttpConnection)Connector.open("http://localhost:8080/TA.html");
                        int length =
(int)hc.getLength();

                        port = hc.getPort();
                        protocol = hc.getProtocol();

```

```

hc.openInputStream();

byte[length];

is.read(data);

IOException("length unknown");

String(data);

StringBuffer();

        sb.append((char)temp);

is.read();

StringItem("TA.html content is : ", result));

StringItem("length : ", String.valueOf(length)));

StringItem("port : ", String.valueOf(port)));

StringItem("host : ", host));

StringItem("protocol : ", protocol));

StringItem("exception:",exception.getMessage()));

```

```

host = hc.getHost();
if(length>0)
{
    is =

    byte data[] = new

    int wereRead =

    if(wereRead!=length)
    {
        throw new

    }
    result = new

}
else
{
    StringBuffer sb = new

    int temp = is.read();
    while(temp!=-1)
    {

        temp =

    }
    result = sb.toString();
}
form.append(new

form.append(new

form.append(new

form.append(new

form.append(new

}
catch(Exception exception)
{
    form.append(new

```

```

        }
        finally
        {
            try
            {
                if(hc!=null)
                {
                    hc.close();
                }
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
            try
            {
                if(is!=null)
                {
                    is.close();
                }
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }
    }
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

;

//setting the listeners
form.setCommandListener(listener);

//getting the display object
display = Display.getDisplay(this);
}

```

```

    public void startApp()
    {
        display.setCurrent(form);
        System.out.println("form was set as the current screen");
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        exitCommand = null;
        retrieveCommand = null;
    }
}

```

The following example presents a midlet that calls a servlet, sends it a parameter and presents the answer. The servelt is invoked using the GET method.

//filename:SimpleHttpDemo1.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
 //All rights reserved. No part of the contents of this program may be
 //reproduced or transmitted in any form or by any means without the
 //written permission of the publisher.

```

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;

public class SimpleHttpDemo1 extends MIDlet
{
    private Display display;
    private Form form;
    private Command exitCommand, retrieveCommand;

```



```

public SimpleHttpDemo1()
{
    //create the backCommand, okCommand and the exitCommand
    exitCommand = new Command("Exit", Command.EXIT, 1);
    retrieveCommand = new Command("Retrieve", Command.SCREEN,
1);

    //creating the form
    form = new Form("HTTPDEMO1");
    form.addCommand(exitCommand);
    form.addCommand(retrieveCommand);

    //instantiating the listener
    CommandListener listener = new CommandListener()
    {
        public void commandAction(Command c,
Displayable d)
        {
            if(c==exitCommand)
            {
                System.out.println("exitCommand was pressed");
                exit();
            }
            else
            {
                if(c==retrieveCommand)
                {
                    System.out.println("retrieveCommand was pressed");

String msgToSend =
"Haim";
form.append(new
StringItem("Send:", msgToSend));

HttpConnection hc =
null;
DataInputStream dis =
null;

                try
                {
                    String result
= null;
                    hc =
(HttpConnection)Connector.open("http://localhost:8080/servlet/HelloTo?towhom="+msg
ToSend,Connector.READ_WRITE);

```

```

hc.openDataInputStream();

sb = new StringBuffer();

    System.out.println("The StringBuffer object was created");
dis.read();

    System.out.println("dis.read() was invoked");
    while(temp!=-1)
    {
        System.out.println("temp="+ (char)temp);
        sb.append((char)temp);
        temp = dis.read();
    }
    result = sb.toString();

    System.out.println("result="+result);

    form.append(new StringItem("Receive:", result));
}
catch(Exception
exception)
{
    exception.printStackTrace();
    form.append(new StringItem("exception:",exception.getMessage()));
}
finally
{
    try
    {
        if(hc!=null)
        {
            hc.close();
        }
    }
}

```

```

    }

    catch(Exception e)
    {

        e.printStackTrace();

    }
    try
    {

        if(dis!=null)
        {
            dis.close();
        }

    }

    catch(Exception e)
    {

        e.printStackTrace();

    }

}

}

}

}

    }

    public void exit()
    {
        destroyApp(true);
        notifyDestroyed();
    }

}

    //setting the listeners
    form.setCommandListener(listener);

    //getting the display object
    display = Display.getDisplay(this);
}

public void startApp()
{
    display.setCurrent(form);
    System.out.println("form was set as the current screen");
}

```

```

    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        exitCommand = null;
        retrieveCommand = null;
    }
}

```

//filename: HelloTo.java

//Copyright (c) 2000 Haim Michael & Zindell Publishing House, Ltd.
 //All rights reserved. No part of the contents of this program may be
 //reproduced or transmitted in any form or by any means without the
 //written permission of the publisher.

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

```

```

public class HelloTo extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse result)
        throws ServletException, IOException
    {
        result.setContentType("text/html");
        PrintWriter out = result.getWriter();
        String toWhom = request.getParameter("towhom");
        out.println("<HTML>");
        out.println("<HEAD><TITLE>midlet-servlet
communication</TITLE></HEAD>");
        out.println("<BODY><H1>Hello to " + toWhom + "</BODY>");
    }
}

```

```
        out.println("</HTML>");
        out.flush();
        out.close();
    }

    public void doPost(HttpServletRequest request, HttpServletResponse result)
        throws ServletException, IOException
    {
        doGet(request, result);
    }
}
```

Chapter 6:

The Canvas

- • [Introduction](#)
- • [Drawing Different Shapes, Text and Images](#)
- • [Low Level Events Handling](#)
- • [Animation Techniques and Double Buffering](#)

Introduction

The Canvas abstract class extends Displayable. The Canvas class provides the paint() method as well as some other methods that overriding them is the way for handling low level events (similar to events handling in Java 1.0 before the emergence of the delegation model). When using the Canvas class we extend it and override paint() (an abstract method).

The actual size of the screen varies from device to device. The screen dimension can be queried by calling the getHeight() and the getWidth() methods.

Since the Canvas class extends Displayable it is possible adding commands to it as well as registering it a CommandListener using the setCommandListener() method.

The declaration of the paint abstract method that was declared in Canvas is:

```
public void paint(Graphics g)
```

The Graphics class in J2ME(CLDS\MIDP) is very similar to the Graphics class in J2SE. It defines the following methods:

```
public void translate(int x, int y)
```

```
public int getTranslateX()
```

```
public int getTranslateY()
```

```
public int getColor()

public int getRedComponent()

public int getGreenComponent()

public int getBlueComponent()

public int getGrayScale()

public void setColor(int red, int green, int blue)

public void setColor(int RGB)

public void setGrayScale(int value)

public Font getFont()

public void setStrokeStyle(int style)

public int getStrokeStyle()

public void setFont(Font font)

public int getClipX()

public int getClipY()

public int getClipWidth()

public int getClipHeight()

public void clipRect(int x, int y, int width, int height)

public void setClip(int x, int y, int width, int height)

public void drawLine(int x1, int y1, int x2, int y2)

public void fillRect(int x, int y, int width, int height)

public void drawRect(int x, int y, int width, int height)
```



```
public void drawRoundRect(int x, int y, int width, int height, int arcWidth,  
int arcHeight)
```

```
public void fillRoundRect(int x, int y, int width, int height, int arcWidth,  
int arcHeight)
```

```
public void fillArc(int x, int y, int width, int height, int startAngle,  
int arcAngle)
```

```
public void drawArc(int x, int y, int width, int height, int startAngle,  
int arcAngle)
```

```
public void drawString(String str, int x, int y, int anchor)
```

```
public void drawSubstring(String str, int offset, int len, int x, int y,  
int anchor)
```

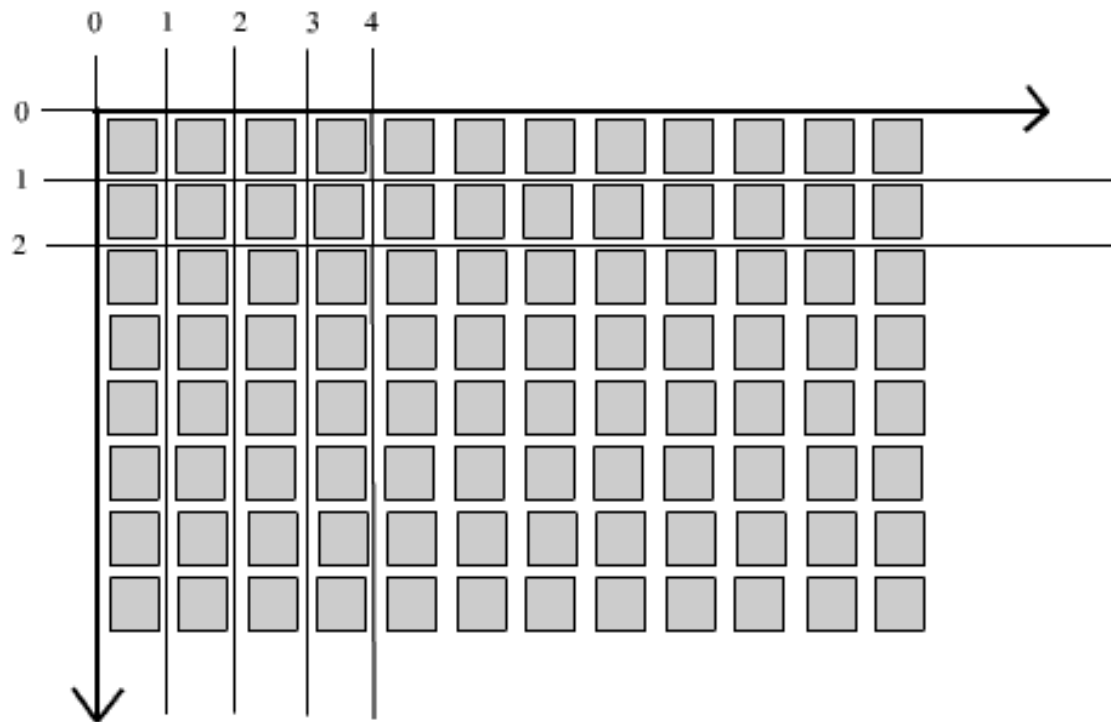
```
public void drawChar(char character, int x, int y, int anchor)
```

```
public void drawChars(char[] data, int offset, int length, int x, int y,  
int anchor)
```

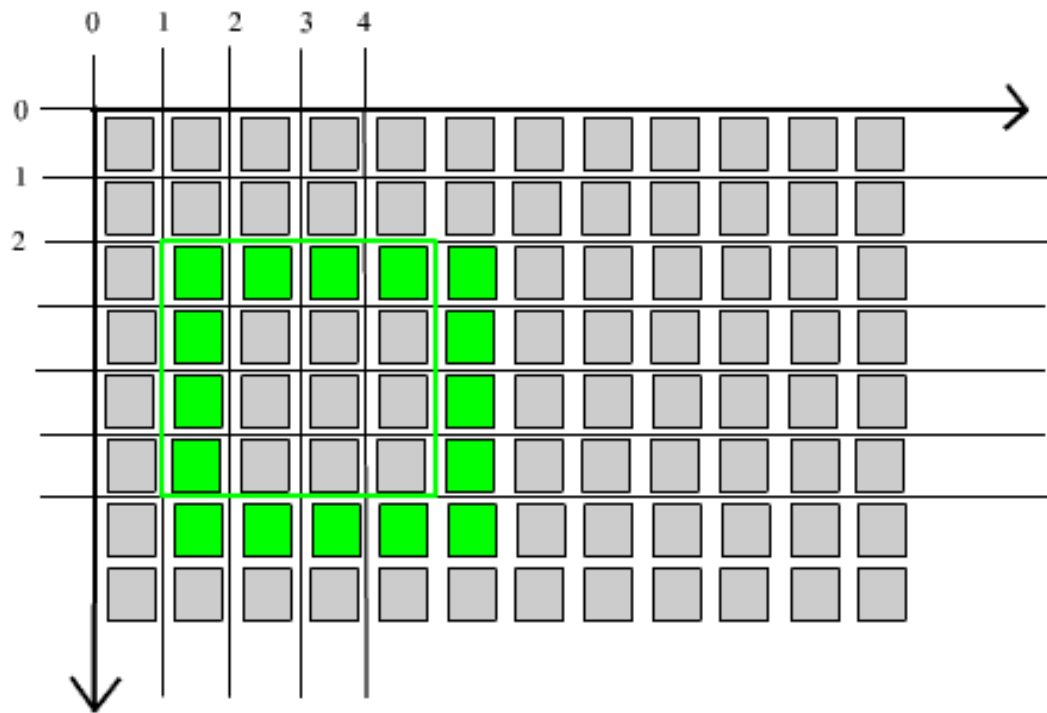
```
public void drawImage(Image img, int x, int y, int anchor)
```

The *Graphics* object can be rendered to the screen or to an off screen image buffer (similar to the *Graphics* class of the J2SE). The midlet can draw onto the *Canvas* object (so it will be drawn to the screen) using the *Graphics* object only during the *paint()* method invocation. Rendering to an offscreen image buffer is done by invoking the *getGraphics()* method on a specific image. Rendering requests on a *Graphics* object that was retrieved by invoking the *getGraphics()* method on a specific image can be done at any time.

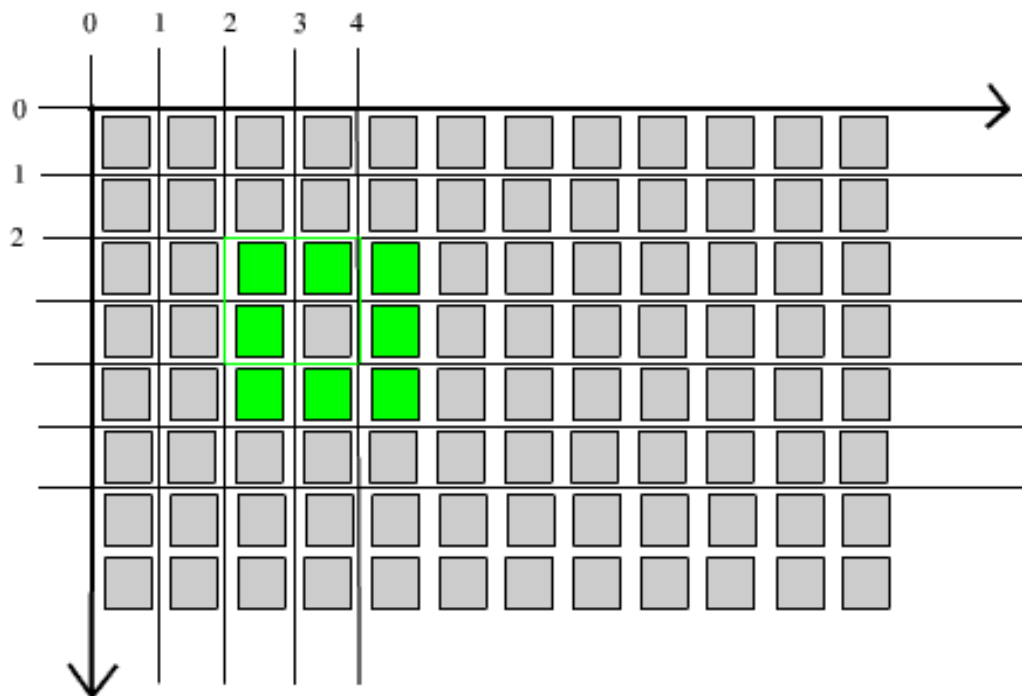
The coordinate system is anchored in the upper-left corner of the drawing surface. The coordinate system represents the locations between the pixels and not the pixels themselves. The following figure demonstrates it:



When drawing operation is done on the *Graphics* object only the pixels to the right and beneath the drawing path will be painted. Operation such as `drawRect(1,2,4,4)` will have the following effect:



And operation such as `drawRect(2,2,2,2)` will have the following effect:



Each Graphics object maintain several attributes that describe the rendering:
color, **font** and **stroke styles**.

Color

Currently, most of the MIDP implementation (devices) usually do not support the use of colors or just support a primitive color model (like 8 bit color model).

Therefore, those MIDP implementation simply map each of the colors that the midlet wants to one of the available colors. Furthermore, the midlet also has the capability to check whether the device support color representation by calling the `Display.isColor()` static method. The midlet can also check the number of colors the device support by calling the `Display.numColors()` static method. If the device

doesn't support different colors the method will return the number of different grey levels. The `Graphics` class has the following color related methods:

```
public int getColor()  
public int getBlueComponent()  
public int getGreenComponent()  
public int getRedComponent()  
public int getGrayScale()  
public void setColor(int RGB)  
public void setColor(int red, int green, int blue)  
public setGrayScale(int value)
```

The following midlet presents a simple color simulation. Note that the midlet finds the device screen dimensions and use it. This way, the midlet portability across different devices improves.

```
//filename:ColorsDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;  
import javax.microedition.io.*;
```

```
public class ColorsDemo extends MIDlet implements CommandListener  
{  
    private Display display;  
    private Form form;  
    private Command exitCommand, startCommand;  
    private MyCanvas myCanvas;
```

```

public ColorsDemo()
{
    //create the backCommand, okCommand and the exitCommand
    exitCommand = new Command("Exit", Command.EXIT, 1);
    startCommand = new Command("Start", Command.OK, 1);

    //creating the canvas
    myCanvas = new MyCanvas();
    myCanvas.addCommand(exitCommand);

    //creating the form
    form = new Form("C O L O R S");
    form.addCommand(exitCommand);
    form.addCommand(startCommand);
    System.out.println("commands were added");

    //setting the listeners
    form.setCommandListener(this);

    //getting the display object
    display = Display.getDisplay(this);
}

public void commandAction(Command c, Displayable d)
{
    System.out.println("within commandAction");
    if(c==exitCommand)
    {
        myCanvas.stopMoving();
        System.out.println("exitCommand was pressed");
        exit();
    }
    else
    {
        if(c==startCommand)
        {
            System.out.println("startCommand was pressed");
            display.setCurrent(myCanvas);
            myCanvas.startMoving();
        }
    }
}

public void exit()
{

```

```

        destroyApp(true);
        notifyDestroyed();
    }

    public void startApp()
    {
        display.setCurrent(form);
        System.out.println("form was set as the current screen");
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        exitCommand = null;
        startCommand = null;
    }
}

```

```

class MyCanvas extends Canvas implements Runnable
{
    private Thread thread;
    private int width;
    private int height;
    private int radius;
    private int x,y;
    private int xStep=1;
    private int yStep=1;
    private boolean goOn;
    private int r,g,b;
    private int rStep=10, gStep=20, bStep=30;

    MyCanvas()
    {
        width = this.getWidth();
        height = this.getHeight();
        radius = Math.min(width,height)/12;
        x = 0;
        y = 0;
    }
}

```

```

void startMoving()
{
    if(thread==null)
    {
        goOn = true;
        thread = new Thread(this);
        thread.start();
    }
}

public void run()
{
    while(goOn)
    {
        if(x>width-radius*2-xStep && xStep>0 || x<xStep &&
xStep<0)
        {
            xStep=-xStep;
        }
        if(y>height-radius*2-yStep && yStep>0 || y<yStep &&
yStep<0)
        {
            yStep=-yStep;
        }
        repaint();
    }
}

public void paint(Graphics graphics)
{
    graphics.setColor(255,255,255);
    graphics.fillRect(0,0,width,height);
    graphics.setColor(r,g,b);
    x+=xStep;
    y+=yStep;
    graphics.fillRoundRect(x,y,2*radius,2*radius,radius,radius);
    rgbIncrease();
}

void rgbIncrease()
{
    if(r>255-rStep && rStep>0 || r<-rStep && rStep<0)
    {
        rStep=-rStep;
    }
}

```



```

        if(g>255-gStep && gStep>0 || g<-rStep && gStep<0)
        {
            gStep=-gStep;
        }
        if(b>255-bStep && bStep>0 || b<-rStep && bStep<0)
        {
            bStep=-bStep;
        }
        r+=rStep;
        g+=rStep;
        b+=rStep;
    }

    void stopMoving()
    {
        if(thread!=null)
        {
            goOn = false;
            thread = null;
        }
    }
}

```

The following midlet is another example of switching between different colors.

//filename:ColorSwitch.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
 //All rights reserved. No part of the contents of this program may be
 //reproduced or transmitted in any form or by any means without the
 //written permission of the publisher.

```

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;

```

```

public class ColorSwitch extends MIDlet implements CommandListener
{
    private Display display;
    private Form form;

```

```

private Command exitCommand, startCommand;
private ColorCanvas colorCanvas;

public ColorSwitch()
{
    //create the backCommand, okCommand and the exitCommand
    exitCommand = new Command("Exit", Command.EXIT, 1);
    startCommand = new Command("Start", Command.OK, 1);

    //creating the canvas
    colorCanvas = new ColorCanvas();
    colorCanvas.addCommand(exitCommand);

    //creating the form
    form = new Form("C O L O R S");
    form.addCommand(exitCommand);
    form.addCommand(startCommand);
    System.out.println("commands were added");

    //setting the listeners
    form.setCommandListener(this);

    //getting the display object
    display = Display.getDisplay(this);
}

public void commandAction(Command c, Displayable d)
{
    System.out.println("within commandAction");
    if(c==exitCommand)
    {
        System.out.println("exitCommand was pressed");
        exit();
    }
    else
    {
        if(c==startCommand)
        {
            System.out.println("startCommand was pressed");
            display.setCurrent(colorCanvas);
        }
    }
}

public void exit()

```

```

    {
        destroyApp(true);
        notifyDestroyed();
    }

    public void startApp()
    {
        display.setCurrent(form);
        System.out.println("form was set as the current screen");
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        exitCommand = null;
        startCommand = null;
    }
}

```

```

class ColorCanvas extends Canvas
{
    private int width;
    private int height;
    private int r=0,g=0,b=100;
    private int rStep, gStep;

    ColorCanvas()
    {
        width = this.getWidth();
        height = this.getHeight();
        rStep = 5*255/width;
        gStep = 5*255/height;
    }

    public void paint(Graphics graphics)
    {
        for(int col=0; col<height; col+=5)
        {
            g+=gStep;

```

```

        rStep = 0;
        for(int row=0; row<width; row+=5)
        {
            r+=rStep;
            graphics.setColor(r,g,b);
            graphics.fillRect(row,col,5,5);
            System.out.println("row="+row+" col="+col);
        }
    }
}

```

By calling the `setGrayscale()` method it is possible to set the gray level (when the device doesn't support color this method become very relevant).

Drawing Different Shapes, Text and Images

The `Canvas` abstract class extends `Displayable`. The `Canvas` class provides the abstract `paint()` method. This method should be overridden in the class you declare as a class that extends `Canvas`.

When we draw lines (stand alone lines or part of a shape) the `Graphics` class provides two strokes styles: `DOTTED` and `SOLID`. The relevant methods in the `Graphics` class are:

```
public int getStrokeStyle()  
public void setStrokeStyle(int style)
```

The following midlet presents these two types of strokes by drawing several lines (DOTTED and SOLID lines). The method to draw a line is: `public void drawLine(int x1, int y1, int x2, int y2)`.

```
//filename:LinesDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;  
import javax.microedition.io.*;
```

```
public class LinesDemo extends MIDlet implements CommandListener  
{  
    private Display display;  
    private Form form;  
    private Command exitCommand, startCommand;  
    private LinesCanvas linesCanvas;  
  
    public LinesDemo()  
    {  
        //create the backCommand, okCommand and the exitCommand  
        exitCommand = new Command("Exit", Command.EXIT, 1);  
        startCommand = new Command("Start", Command.OK, 1);  
  
        //creating the canvas  
        linesCanvas = new LinesCanvas();  
        linesCanvas.addCommand(exitCommand);  
  
        //creating the form  
        form = new Form("L I N E S");  
        form.addCommand(exitCommand);  
        form.addCommand(startCommand);  
        System.out.println("commands were added");  
    }  
}
```

```

        //setting the listeners
        form.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void commandAction(Command c, Displayable d)
    {
        System.out.println("within commandAction");
        if(c==exitCommand)
        {
            System.out.println("exitCommand was pressed");
            exit();
        }
        else
        {
            if(c==startCommand)
            {
                System.out.println("startCommand was pressed");
                display.setCurrent(linesCanvas);
            }
        }
    }

    public void exit()
    {
        destroyApp(true);
        notifyDestroyed();
    }

    public void startApp()
    {
        display.setCurrent(form);
        System.out.println("form was set as the current screen");
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
    }

```

```

        form = null;
        exitCommand = null;
        startCommand = null;
        linesCanvas = null;
    }
}

class LinesCanvas extends Canvas
{
    private int width;
    private int height;

    LinesCanvas()
    {
        width = this.getWidth();
        height = this.getHeight();
    }

    public void paint(Graphics graphics)
    {
        //drawing the axes
        graphics.setColor(0x8A2BE2); //blue
        graphics.drawLine(0,height/2,width,height/2);
        graphics.drawLine(width,height/2,width-10,height/2-10);
        graphics.drawLine(width,height/2,width-10,height/2+10);
        graphics.drawLine(width/2,height,width/2,0);
        graphics.drawLine(width/2,0,width/2-10,10);
        graphics.drawLine(width/2,0,width/2+10,10);

        //drawing a dotted line
        graphics.setColor(0xFF0000); //red
        graphics.setStrokeStyle(Graphics.DOTTED);
        graphics.drawLine(10,height-10,width-10,10);
    }
}

```

The *Graphics* class provides different methods for drawing/filling rectangles:

public void drawRect(int x, int y, int width, int height)

public void fillRect(int x, int y, int width, int height)

public void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)

```
public void fillRoundRect(int x, int y, int width, int height, int arcWidth, int  
arcHeight)
```

The following midlet allows the user getting a bar chart calculated based on two numbers.

```
//filename:GraphsDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;  
import javax.microedition.io.*;
```

```
public class GraphsDemo extends MIDlet implements CommandListener  
{  
    private Display display;  
    private Form form;  
    private Command exitCommand, startCommand, backCommand;  
    private GraphCanvas graphCanvas;  
    private TextField tf1, tf2;  
  
    public GraphsDemo()  
    {  
        //create the commands  
        exitCommand = new Command("Exit", Command.EXIT, 1);  
        startCommand = new Command("Start", Command.SCREEN, 1);  
        backCommand = new Command("Back", Command.BACK, 1);  
  
        //creating the canvas  
        graphCanvas = new GraphCanvas();  
        graphCanvas.addCommand(startCommand);  
        graphCanvas.addCommand(backCommand);  
  
        //creating the form  
        form = new Form("GRAPHS");  
    }  
}
```



```

        form.addCommand(exitCommand);
        form.addCommand(startCommand);
        System.out.println("commands were added");
        tf1 = new TextField("num.1:", "10", 6, TextField.NUMERIC);
        tf2 = new TextField("num.2:", "10", 6, TextField.NUMERIC);
        form.append(tf1);
        form.append(tf2);

        //setting the listeners
        form.setCommandListener(this);
        graphCanvas.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void commandAction(Command c, Displayable d)
    {
        System.out.println("within commandAction");
        if(c==exitCommand)
        {
            System.out.println("exitCommand was pressed");
            exit();
        }
        else
        {
            if(c==startCommand)
            {
                System.out.println("startCommand was pressed");
                int num1 = Integer.parseInt(tf1.getString());
                int num2 = Integer.parseInt(tf2.getString());
                graphCanvas.setNumbers(num1,num2);
                display.setCurrent(graphCanvas);
            }
            else
            {
                if(c==backCommand)
                {
                    display.setCurrent(form);
                }
            }
        }
    }

    public void exit()
    {

```

```

        destroyApp(true);
        notifyDestroyed();
    }

    public void startApp()
    {
        display.setCurrent(form);
        System.out.println("form was set as the current screen");
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        exitCommand = null;
        startCommand = null;
        graphCanvas = null;
    }
}

```

```

class GraphCanvas extends Canvas
{
    private int width;
    private int height;
    private int num1, num2, scale, rate;

    GraphCanvas()
    {
        width = this.getWidth();
        height = this.getHeight();
    }

    void setNumbers(int num1, int num2)
    {
        this.num1 = num1;
        this.num2 = num2;
    }

    public void paint(Graphics graphics)
    {

```

```

        int widthCell = width/8;
        graphics.setColor(0,0,0);
        graphics.drawLine(0,height-10,width,height-10);
        graphics.setColor(0x008000);
        graphics.fillRect(widthCell, height-10-num1,widthCell*2,num1);
        graphics.setColor(0xFFFF00);
        graphics.fillRect(widthCell*5,height-10-num2,widthCell*2,num2);
    }
}

```

The *Graphics* class provides methods for drawing\filling circular shapes:

```
public void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)
```

```
public void fillArc(int x, int y, int width, int height, int startAngel, int arcAngle)
```

The following midlet draw a PIE chart according to three numbers it get.

```
//filename:PIEDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;
```

```
public class PIEDemo extends MIDlet implements CommandListener
{
    private Display display;
    private Form form;
    private Command exitCommand, startCommand, backCommand;
    private PIECanvas pieCanvas;
    private TextField tf1, tf2, tf3;

    public PIEDemo()
    {
        //create the commands
    }
}

```

```

        exitCommand = new Command("Exit", Command.EXIT, 1);
        startCommand = new Command("Start", Command.SCREEN, 1);
        backCommand = new Command("Back", Command.BACK, 1);

        //creating the canvas
        pieCanvas = new PIECanvas();
        pieCanvas.addCommand(startCommand);
        pieCanvas.addCommand(backCommand);

        //creating the form
        form = new Form("GRAPHS");
        form.addCommand(exitCommand);
        form.addCommand(startCommand);
        System.out.println("commands were added");
        tf1 = new TextField("num.1:", "10", 6, TextField.NUMERIC);
        tf2 = new TextField("num.2:", "20", 6, TextField.NUMERIC);
        tf3 = new TextField("num.3:", "30", 6, TextField.NUMERIC);
        form.append(tf1);
        form.append(tf2);
        form.append(tf3);

        //setting the listeners
        form.setCommandListener(this);
        pieCanvas.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void commandAction(Command c, Displayable d)
    {
        System.out.println("within commandAction");
        if(c==exitCommand)
        {
            System.out.println("exitCommand was pressed");
            exit();
        }
        else
        {
            if(c==startCommand)
            {
                System.out.println("startCommand was pressed");
                int num1 = Integer.parseInt(tf1.getString());
                int num2 = Integer.parseInt(tf2.getString());
                int num3 = Integer.parseInt(tf3.getString());
                pieCanvas.setNumbers(num1,num2,num3);
            }
        }
    }

```

```

        display.setCurrent(pieCanvas);
    }
    else
    {
        if(c==backCommand)
        {
            display.setCurrent(form);
        }
    }
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void startApp()
{
    display.setCurrent(form);
    System.out.println("form was set as the current screen");
}

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    display = null;
    form = null;
    exitCommand = null;
    startCommand = null;
    pieCanvas = null;
}

}

class PIECanvas extends Canvas
{
    private int width;
    private int height;
    private int num1, num2, num3;

```

```

PIECanvas()
{
    width = this.getWidth();
    height = this.getHeight();
}

void setNumbers(int num1, int num2, int num3)
{
    this.num1 = num1;
    this.num2 = num2;
    this.num3 = num3;
}

public void paint(Graphics graphics)
{
    int bigNum1, bigNum2, bigNum3, sum;
    bigNum1 = num1 * 1000;
    bigNum2 = num2 * 1000;
    bigNum3 = num3 * 1000;
    sum = bigNum1 + bigNum2 + bigNum3;
    int rate = sum / 360;
    int angel1, angel2, angel3;
    angel1 = bigNum1 / rate;
    angel2 = bigNum2 / rate;
    angel3 = 360 - angel1 - angel2;

    //painting the PIE chart
    graphics.setColor(0,0,0);
    graphics.fillArc(5,5,width-10,height-10,0,angel1);
    graphics.setColor(0x008000);
    graphics.fillArc(5,5,width-10,height-10,angel1,angel1+angel2);
    graphics.setColor(0xFFFF00);
    graphics.fillArc(5,5,width-10,height-10,angel1+angel2, 360-angel1-
angel2);
}
}

```

The *Graphics* class provides several methods for drawing text to the screen:

```

public void drawChar(char character, int x, int y, int ancor)
public void drawChars(char []data, int offset, int length, int x, int y, int
ancor)

```

```
public void drawstring(String str, int x, int y, int anchor)
```

```
public void drawSubstring(String str, int offset, int len, int x, int y, in ancor)
```

The Graphics object has the font attribute that holds a reference to Font object.

The font object represents specific font and font metrics. You can set the font attribute by calling the setFont() method:

```
public void setFont(Font font)
```

The Font class provides the following method:

```
public static Font getFont(int face, int style, int size)
```

This method obtains a Font object according to the attributes it receives. If a matching font does not exist then the closest match font will be returned. The values you can send to this method are final values the Font class has:

face - FACE_SYSTEM, FACE_MONOSPACE or FACE_PROPORTIONAL

style - STAYLE_PLAIN, STYLE_BOLD, STYLE_ITALIC, STYLE_UNDERLINED or combination of several of these values using the | operator.

size - SIZE_SMALL, SIZE_MEDIUM or SIZE_LARGE

In order to get the default device font, invoke the following static method (declared in the Font class):

```
public static Font getDefaultFont()
```

When you draw a string it is measured in the following way:



Once you have a Font object, different method enables you receiving its attributes as well as more information.

public int getFace()

This method returns the one of the possible font faces that best describes the given font (according to the final values that were declared in the Font class): Font.FACE_SYSTEM, Font.FACE_MONOSPACE or Font.FACE_PROPORTIONAL.

public int getHeight()

This method returns the font height (the font height as described in the above diagram).

public int getSize()

This method returns the font size. The returned value is one of the final static values the Font class declare: SIZE_SMALL, SIZE_MEDIUM or SIZE_LARGE.

public int getStyle()

This method returns the style of the font which is one (or | combination) of the following values: `STYLE_BOLD`, `STYLE_ITALIC`, `STYLE_UNDERLINED` or `STYLE_PLAIN`.

`public boolean isBold()`

This method returns a boolean value that indicates whether the font is bold.

`public boolean isItalic()`

This method returns a boolean value that indicates whether the font is italic.

`public boolean isUnderlined()`

This method returns a boolean value that indicates whether the font is underlined.

`public int getBaselinePosition()`

This method returns the distance in pixels from the top of the text to its baseline.

`public int charWidth(char ch)`

This method returns the advance width of the specific character it receives (the amount by which the current point is moved from one character to the next in the same base line including the char space itself).

`public int charsWidth(char []vec, int offset, int length)`

This method returns the advance width of the characters in the array it receives. `offset` - is the index of the first character to measure. `length` - is the number of characters to measure.

`public int stringWidth(String str)`

This method returns the total advance width for showing `str`. It returns the string width as showed in the last diagram.

`public int substringWidth(String str, int offset, int len)`

This method returns the total advance width for showing the specified substring in this font. It returns the substring width.

Once you have your desired Font object, you can set it in the Graphics object you have by calling the `setFont` method and starts rendering text to the screen.

```
public void setFont(Font font)
```

The different methods the *Graphics* class provides for drawing text are:

```
public void drawChar(char character, int x, int y, int ancor)
```

```
public void drawChars(char []data, int offset, int length, int x, int y, int  
ancor)
```

```
public void drawstring(String str, int x, int y, int anchor)
```

```
public void drawSubstring(String str, int offset, int len, int x, int y, in ancor)
```

These methods not only get the location (x,y) in which the text will be drawn. These methods also receive the ancor value that indicates the position of (x,y) in relation to the drawn text. The ancor value is a combined integer value (using the OR operator) of two values: one that indicates the horizontal position and another that indicates the vertical position. The horizontal value can be one of the following values: *Graphics.LEFT*, *Graphics.RIGHT* and *Graphics.HCENTER*. The vertical value can be one of the following values: *Graphics.TOP*, *Graphics.BASELINE* and *Graphics.BOTTOM*.

The following midlet presents a simple example of drawing text to the screen.

```
//filename:TextDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;  
import javax.microedition.io.*;
```

```

public class TextDemo extends MIDlet implements CommandListener
{
    private Display display;
    private Form form;
    private Command exitCommand, backCommand, startCommand;
    private TextCanvas textCanvas;

    public TextDemo()
    {
        //create the commands
        exitCommand = new Command("Exit", Command.EXIT, 1);
        backCommand = new Command("Back", Command.BACK, 1);
        startCommand = new Command("Start", Command.OK, 1);

        //creating the canvas
        textCanvas = new TextCanvas();
        textCanvas.addCommand(backCommand);

        //creating the form
        form = new Form("TEXT DEMO");
        form.addCommand(exitCommand);
        form.addCommand(startCommand);
        System.out.println("commands were added");

        //setting the listeners
        form.setCommandListener(this);
        textCanvas.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void commandAction(Command c, Displayable d)
    {
        System.out.println("within commandAction");
        if(c==exitCommand)
        {
            System.out.println("exitCommand was pressed");
            exit();
        }
        else
        {
            if(c==startCommand)
            {
                System.out.println("startCommand was pressed");
                display.setCurrent(textCanvas);
            }
        }
    }
}

```

```

        }
        else
        {
            if(c==backCommand)
            {
                display.setCurrent(form);
            }
        }
    }

    public void exit()
    {
        destroyApp(true);
        notifyDestroyed();
    }

    public void startApp()
    {
        display.setCurrent(form);
        System.out.println("form was set as the current screen");
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        exitCommand = null;
        backCommand = null;
        startCommand = null;
        textCanvas = null;
    }
}

class TextCanvas extends Canvas
{
    private int width;
    private int height;
    private int num1, num2, num3;

```

```

    TextCanvas()
    {
        width = this.getWidth();
        height = this.getHeight();
    }

    public void paint(Graphics graphics)
    {
        graphics.setColor(0x008000);
        graphics.setFont(Font.getFont(Font.FACE_SYSTEM,
Font.STYLE_PLAIN, Font.SIZE_SMALL));
        graphics.drawString("ZINDELL", 10, 30, Graphics.BOTTOM |
Graphics.LEFT);
        graphics.setColor(0x0080FF);
        graphics.setFont(Font.getFont(Font.FACE_SYSTEM,
Font.STYLE_PLAIN, Font.SIZE_MEDIUM));
        graphics.drawString("ZINDELL", 10, 45, Graphics.BOTTOM |
Graphics.LEFT);
        graphics.setColor(0xEE92EE);
        graphics.setFont(Font.getFont(Font.FACE_SYSTEM,
Font.STYLE_PLAIN, Font.SIZE_LARGE));
        graphics.drawString("ZINDELL", 10, 60, Graphics.BOTTOM |
Graphics.LEFT);
    }
}

```

The *Graphics* class provides a method for drawing an image:

public void drawImage(Image image, int x, int y, int anchor)

The image you can draw using this method doesn't have to be immutable. It can also be mutable. When giving an anchor value, instead of using the *Graphics.BASELINE* you should use the *Graphics.VCENTER* and instead of using the *Graphics*.

The following midlet present a simple image drawing.

//filename:ImageDemo.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
//All rights reserved. No part of the contents of this program may be

//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;
```

```
public class ImageDemo extends MIDlet implements CommandListener
{
    private Display display;
    private Form form;
    private Command exitCommand, backCommand, startCommand;
    private ImageCanvas imageCanvas;

    public ImageDemo()
    {
        //create the commands
        exitCommand = new Command("Exit", Command.EXIT, 1);
        backCommand = new Command("Back", Command.BACK, 1);
        startCommand = new Command("Start", Command.OK, 1);

        //creating the canvas
        imageCanvas = new ImageCanvas();
        imageCanvas.addCommand(backCommand);
        System.out.println("The image canvas was created");

        //creating the form
        form = new Form("TEXT DEMO");
        form.addCommand(exitCommand);
        form.addCommand(startCommand);
        System.out.println("commands were added");

        //setting the listeners
        form.setCommandListener(this);
        imageCanvas.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void commandAction(Command c, Displayable d)
    {
        System.out.println("within commandAction");
    }
}
```

```

        if(c==exitCommand)
        {
            System.out.println("exitCommand was pressed");
            exit();
        }
        else
        {
            if(c==startCommand)
            {
                System.out.println("startCommand was pressed");
                display.setCurrent(imageCanvas);
            }
            else
            {
                if(c==backCommand)
                {
                    display.setCurrent(form);
                }
            }
        }
    }

    public void exit()
    {
        destroyApp(true);
        notifyDestroyed();
    }

    public void startApp()
    {
        display.setCurrent(form);
        System.out.println("form was set as the current screen");
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        exitCommand = null;
        backCommand = null;
        startCommand = null;
        imageCanvas = null;
    }

```

```

    }
}

class ImageCanvas extends Canvas
{
    private int width;
    private int height;
    private Image img;

    ImageCanvas()
    {
        width = this.getWidth();
        height = this.getHeight();
        try
        {
            img = Image.createImage("/house.png");
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }

    public void paint(Graphics graphics)
    {
        graphics.drawImage(img,width/2,height/2,graphics.HCENTER|graphics.VCENT
ER);
    }
}

```

The translate method:

public void translate(int x, int y)

enables translating the origin of the coordinate system. It simply moves the origin to point (x,y) of the current coordinate system. The coordinates of the translated origin can be retrieved using:

public int getTranslateX()

public int getTranslateY()

These two methods return the x and y translate values relative to the default coordinate system.

The following midlet presents a simple use of the translate method. More useful use of this method can be in the creation of games and in showing images in which the picture to be displayed is bigger than the device screen. Using the translate method it is possible leaving the user to decide which parts of the whole picture he wants to see.

```
//filename:TranslateDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;  
import javax.microedition.io.*;
```

```
public class TranslateDemo extends MIDlet implements CommandListener  
{  
    private Display display;  
    private Form form;  
    private Command exitCommand, backCommand, startCommand;  
    private TranslateCanvas translateCanvas;  
  
    public TranslateDemo()  
    {  
        //create the commands  
        exitCommand = new Command("Exit", Command.EXIT, 1);  
        backCommand = new Command("Back", Command.BACK, 1);  
        startCommand = new Command("Start", Command.OK, 1);  
  
        //creating the canvas  
        translateCanvas = new TranslateCanvas();  
    }  
}
```

```

        translateCanvas.addCommand(backCommand);
        System.out.println("The image canvas was created");

        //creating the form
        form = new Form("TRANSLATE DEMO");
        form.addCommand(exitCommand);
        form.addCommand(startCommand);
        System.out.println("commands were added");

        //setting the listeners
        form.setCommandListener(this);
        translateCanvas.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void commandAction(Command c, Displayable d)
    {
        System.out.println("within commandAction");
        if(c==exitCommand)
        {
            System.out.println("exitCommand was pressed");
            exit();
        }
        else
        {
            if(c==startCommand)
            {
                System.out.println("startCommand was pressed");
                display.setCurrent(translateCanvas);
            }
            else
            {
                if(c==backCommand)
                {
                    display.setCurrent(form);
                }
            }
        }
    }

    public void exit()
    {
        destroyApp(true);
        notifyDestroyed();
    }

```

```

    }

    public void startApp()
    {
        display.setCurrent(form);
        System.out.println("form was set as the current screen");
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        exitCommand = null;
        backCommand = null;
        startCommand = null;
        translateCanvas = null;
    }
}

```

```

class TranslateCanvas extends Canvas
{
    private int width;
    private int height;
    private Image img;

    TranslateCanvas()
    {
        width = this.getWidth();
        height = this.getHeight();
        try
        {
            img = Image.createImage("/smallBird.png");
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }

    public void paint(Graphics g)

```

```

    {
        for(int i=0; i<3; i++)
        {

            g.drawImage(img,width/4,height/2,Graphics.HCENTER|Graphics.VCENTER);
            g.translate(20,5);

        }
    }
}

```

The *Graphics* class allows you setting up a working area (clip rectangle) in the canvas. When a working area is set, only the pixels that lie entirely within the clip rectangle are affected by the graphics operations. Pixels outside the clip rectangle are not changed when different graphics methods are called. The *Graphics*' method with which you define a clip is:

public void setClip(int x, int y, int width, int height)

The x and y are the coordinates of the clip rectangle. The width and height are its dimensions.

The *Graphics*' class also enables you intersecting the current clip rectangle with another rectangle. This can be done using the following method:

public void clipRect(int x, int y, int width, int height)

The attributes of the current rectangle clip can be retrieved using the following methods:

public int getClipHeight()

public int getClipWidth()

public int getClipX()

public int getClipY()

The following midlet presents a simple animation using the clip rectangle.

//filename:ClipDemo.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;
import java.util.*;
```

```
public class ClipDemo extends MIDlet implements CommandListener
{
    private Display display;
    private Form form;
    private Command exitCommand, backCommand, startCommand;
    private ClipCanvas clipCanvas;

    public ClipDemo()
    {
        //create the commands
        exitCommand = new Command("Exit", Command.EXIT, 1);
        backCommand = new Command("Back", Command.BACK, 1);
        startCommand = new Command("Start", Command.OK, 1);

        //creating the canvas
        clipCanvas = new ClipCanvas();
        clipCanvas.addCommand(backCommand);
        System.out.println("The image canvas was created");

        //creating the form
        form = new Form("CLIP DEMO");
        form.addCommand(exitCommand);
        form.addCommand(startCommand);
        System.out.println("commands were added");

        //setting the listeners
        form.setCommandListener(this);
        clipCanvas.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
    }
}
```

```

}

public void commandAction(Command c, Displayable d)
{
    System.out.println("within commandAction");
    if(c==exitCommand)
    {
        System.out.println("exitCommand was pressed");
        exit();
    }
    else
    {
        if(c==startCommand)
        {
            System.out.println("startCommand was pressed");
            display.setCurrent(clipCanvas);
            clipCanvas.startAnimation();
        }
        else
        {
            if(c==backCommand)
            {
                clipCanvas.stopAnimation();
                display.setCurrent(form);
            }
        }
    }
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void startApp()
{
    display.setCurrent(form);
    System.out.println("form was set as the current screen");
}

public void pauseApp()
{
}

public void destroyApp(boolean cond)

```

```

    {
        display = null;
        form = null;
        exitCommand = null;
        backCommand = null;
        startCommand = null;
        clipCanvas = null;
    }
}

```

class ClipCanvas extends Canvas implements Runnable

```

{
    private int width;
    private int height;
    private int colors[] = {      0X000000, 0XC0C0C0, 0X808080, 0XFFFFFF,
                                0X008000, 0X00FF00, 0XFF0000, 0X0000FF,
                                0X808080, 0X808000, 0X800080, 0X808000,
                                0XFFFFFF, 0XFFFF00, 0XFF00FF, 0X00FFFF};

    private int colorCounter;
    private Random random;
    private Thread thread;
    private boolean goOn;

    ClipCanvas()
    {
        width = this.getWidth();
        height = this.getHeight();
        random = new Random();
    }

    void startAnimation()
    {
        if(thread==null)
        {
            thread = new Thread(this);
            goOn = true;
            thread.start();
        }
    }

    void stopAnimation()
    {
        if(thread!=null)
        {

```

```

        goOn = false;
        thread = null;
    }
}

public void run()
{
    while(goOn)
    {
        repaint();
    }
}

public void paint(Graphics g)
{
    System.out.println("within the paint method");
    int theColorIndex = Math.abs(random.nextInt()%15);
    System.out.println("theColorIndex="+theColorIndex);
    int theColor = colors[theColorIndex];
    System.out.println("theColor="+theColor);
    g.setColor(theColor);
    System.out.println("color was set");
    int x = random.nextInt()%width;
    int y = random.nextInt()%height;
    System.out.println("x="+x+" y="+y);
    int clipWidth = Math.min(Math.abs(random.nextInt()%64),width-x);
    int clipHeight = Math.min(Math.abs(random.nextInt()%64),height-y);
    g.setClip(x,y,clipWidth,clipHeight);
    g.fillRect(0,0,width,height);
}
}

```

Low level Events Handling

Unlike the high level events handling model which is a delegation model (was introduced in chapter 3), the low level events handling model looks similar to the

JDK1.0 events model. The Canvas class defines several methods that deliver the low level event to the midlet for processing. These methods are called the **event delivery methods** and they include the following:

protected void showNotify()

protected void hideNotify()

protected void keyPressed(int keyCode)

protected void keyReleased(int keyCode)

protected void keyRepeated(int keyCode)

protected void pointerPressed(int x, int y)

protected void pointerReleased(int x, int y)

protected void pointerDragged(int x, int y)

protected abstract void paint(Graphics g)

Each of these methods is serially called by the MIDP implementation when the related event happens and on the same midlet running thread. Thus, when overriding these method you should take care not developing heavy methods. Any of these methods invocations block the midlet running until it returns. Note that the `commandAction()` method you know from the high level of events handling is also called serially on the same midlet running thread.

Besides the paint methods, all the others have empty implementation. Therefore, you must override the paint method, but you don't have to override the others. This way, you will override only the methods that relate to the events you are interested in handling them.

The `showNotify()` method is called right before the canvas become visible on the display. The `hideNotify()` method is called right after the canvas is removed from the display. These methods are called by the MIDP implementation, and it isn't

under the control of the midlet. All the other methods (including the `commandAction` you learned in chapter 3) won't be called if the canvas isn't visible. This means, that the `showNotify()` method must be called before events handling is possible, and once the `hideNotify()` is called neither of the event delivery methods will be called.

When the user presses one of the keys on the keypad, the key event happens. Every key has a "key code" value. MID profile defines the following key codes as static final variables in the `Canvas` class:

```
public static final int KEY_STAR
public static final int KEY_POUND
public static final int KEY_0
public static final int KEY_1
public static final int KEY_2
public static final int KEY_3
public static final int KEY_4
public static final int KEY_5
public static final int KEY_6
public static final int KEY_7
public static final int KEY_8
public static final int KEY_9
```

The string physically printed on the key (on the device) can be retrieved using the following method (was declared in the `Canvas` class):

```
public String getKeyName(int keyCode)
```

This method is very useful when you want to use the physically names that the device has near each one of its keys. This can be very useful when creating a help menu for the user.

The Canvas class has three methods for key events handling:

protected void keyPressed(int keyCode)

This method is invoked whenever the user presses a key.

protected void keyReleased(int keyCode)

This method is invoked whenever the user releases a pressed key.

protected void keyRepeated(int keyCode)

This method is invoked whenever the user presses the same key consecutively more than once in a short time. Not all of the devices support the key-repeated event.

Calling the following method can do checking whether the device supports the key-repeated event:

public boolean hasRepeatedEvents()

The following midlet presents a simple key events handling.

```
//filename:KeyEventsHandling.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;  
import javax.microedition.io.*;  
import java.util.*;
```

```
public class KeyEventsHandling extends MIDlet implements CommandListener  
{  
    private Display display;  
    private Form form;  
    private Command exitCommand, backCommand, startCommand;  
    private KeyEventsCanvas keyEventsCanvas;
```

```

public KeyEventsHandling()
{
    //create the commands
    exitCommand = new Command("Exit", Command.EXIT, 1);
    backCommand = new Command("Back", Command.BACK, 1);
    startCommand = new Command("Start", Command.OK, 1);

    //creating the canvas
    keyEventsCanvas = new KeyEventsCanvas();
    keyEventsCanvas.addCommand(backCommand);

    //creating the form
    form = new Form("KEY EVENTS");
    form.addCommand(exitCommand);
    form.addCommand(startCommand);
    System.out.println("commands were added");

    //setting the listeners
    form.setCommandListener(this);
    keyEventsCanvas.setCommandListener(this);

    //getting the display object
    display = Display.getDisplay(this);
}

public void commandAction(Command c, Displayable d)
{
    System.out.println("within commandAction");
    if(c==exitCommand)
    {
        System.out.println("exitCommand was pressed");
        exit();
    }
    else
    {
        if(c==startCommand)
        {
            System.out.println("startCommand was pressed");
            display.setCurrent(keyEventsCanvas);
        }
        else
        {
            if(c==backCommand)
            {
                display.setCurrent(form);
            }
        }
    }
}

```

```

        }
    }
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void startApp()
{
    display.setCurrent(form);
    System.out.println("form was set as the current screen");
}

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    display = null;
    form = null;
    exitCommand = null;
    backCommand = null;
    startCommand = null;
    keyEventsCanvas = null;
}

}

class KeyEventsCanvas extends Canvas
{
    private int width;
    private int height;
    private int colors[] = { 0X000000, 0XC0C0C0, 0X808080, 0XCC00AA,
                                0X8000, 0X00FF00, 0XFF0000,
0X0000FF,
                                0X808080, 0X808000, 0X800080,
0X008080,
                                0XFFC0CB, 0XFFFF00,
0XFF00FF, 0X00FFFF    };
    private int colorCounter;

```

```

private Random random;
private Font font;
private String msg;
private int fontHeight;

KeyEventsCanvas()
{
    width = this.getWidth();
    height = this.getHeight();
    random = new Random();
    font = Font.getFont(Font.FACE_PROPORTIONAL,
Font.STYLE_BOLD, Font.SIZE_SMALL);
    fontHeight = font.getHeight();
    msg = new String("Press a key ...");
}

protected void keyPressed(int keyCode)
{
    msg = getKeyName(keyCode) + " was pressed";
    repaint();
}

protected void keyReleased(int keyCode)
{
    msg = getKeyName(keyCode) + " was released";
    repaint();
}

public void paint(Graphics g)
{
    System.out.println("within the paint method");
    g.setColor(0xFFFFFFFF);
    g.fillRect(0,0,width,height);
    int theColorIndex = Math.abs(random.nextInt()%15);
    System.out.println("theColorIndex="+theColorIndex);
    int theColor = colors[theColorIndex];
    System.out.println("theColor="+theColor);
    g.setColor(theColor);
    System.out.println("color was set");
    g.setFont(font);
    int msgLength = font.stringWidth(msg);
    g.drawString(msg,(width-msgLength)/2,
        (height-font.getHeight())/2,Graphics.LEFT|Graphics.BASELINE);
}
}

```

The MIDP defines several game actions:

public static final int GAME_A

public static final int GAME_B

public static final int GAME_C

public static final int GAME_D

public static final int UP

public static final int DOWN

public static final int LEFT

public static final int RIGHT

public static final int FIRE

Every key code (one of those that were presented before) can be mapped to at most one game action. However, game action can be associated with more than one key code. This means that in a given MIDP implementation it is possible that moving right shell be done by pressing the right arrow key or by pressing another key such as the "2" button. Each MIDP implementation maps the key codes and the game actions in its unique way. The Canvas class provides the following methods with which you can retrieve the key code that is associated with a game action as well as the game action that is mapped to specific key code.

public int getKeyCode(int gameAction)

public int getGameAction(int keyCode)

Using these methods improves the midlet portability. Note that a game action can have more than one associated key code.

The following midlet presents a simple key events handling. Note the efficient use of the repaint method that receives four arguments, which use for setting the clip area.

```
//filename:GameActionDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;  
import javax.microedition.io.*;  
import java.util.*;
```

```
public class GameActionDemo extends MIDlet implements CommandListener  
{  
    private Display display;  
    private Form form;  
    private Command exitCommand, backCommand, startCommand;  
    private GameActionCanvas gameActionCanvas;  
  
    public GameActionDemo()  
    {  
        //create the commands  
        exitCommand = new Command("Exit", Command.EXIT, 1);  
        backCommand = new Command("Back", Command.BACK, 1);  
        startCommand = new Command("Start", Command.OK, 1);  
  
        //creating the canvas
```



```

gameActionCanvas = new GameActionCanvas();
gameActionCanvas.addCommand(backCommand);

//creating the form
form = new Form("GAMEACTIONS");
form.addCommand(exitCommand);
form.addCommand(startCommand);
System.out.println("commands were added");

//setting the listeners
form.setCommandListener(this);
gameActionCanvas.setCommandListener(this);

//getting the display object
display = Display.getDisplay(this);
}

public void commandAction(Command c, Displayable d)
{
    System.out.println("within commandAction");
    if(c==exitCommand)
    {
        System.out.println("exitCommand was pressed");
        exit();
    }
    else
    {
        if(c==startCommand)
        {
            System.out.println("startCommand was pressed");
            display.setCurrent(gameActionCanvas);
        }
        else
        {
            if(c==backCommand)
            {
                display.setCurrent(form);
            }
        }
    }
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

```

```

    }

    public void startApp()
    {
        display.setCurrent(form);
        System.out.println("form was set as the current screen");
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        exitCommand = null;
        backCommand = null;
        startCommand = null;
        gameActionCanvas = null;
    }
}

```

```

class GameActionCanvas extends Canvas
{
    private int width;
    private int height;
    private Random random;
    private int fontHeight;
    private Image img;
    private int imgWidth;
    private int imgHeight;
    private int x,y;

    GameActionCanvas()
    {
        width = this.getWidth();
        try
        {
            img = Image.createImage("/bird.png");
        }
        catch(Exception exception)
        {
            exception.printStackTrace();
        }
    }
}

```

```

    }
    width = getWidth();
    height = getHeight();
    x = width/2;
    y = height/2;
    imgHeight = img.getHeight()+2;
    imgWidth = img.getWidth()+2;
}

protected void keyPressed(int keyCode)
{
    //handling the game action
    System.out.println("within the keyPressed");
    switch(getGameAction(keyCode))
    {
        case Canvas.DOWN:
            y++;
            break;
        case Canvas.UP:
            y--;
            break;
        case Canvas.LEFT:
            x--;
            break;
        case Canvas.RIGHT:
            x++;
            break;
    }
    repaint(x-imgWidth/2,y-imgHeight/2,imgWidth,imgHeight);
}

public void paint(Graphics g)
{
    System.out.println("within the paint method");
    g.setColor(0);
    g.fillRect(0,0,width,height);
    g.drawImage(img, x, y, Graphics.VCENTER|Graphics.HCENTER);
}
}

```

Some devices (such as the known palm pilot  or other touch screen devices)

support the pointer events, which occur as the user press on specific place of the

screen. The device mechanism that enables the user touching specific part of the screen is mostly the stylus. The Canvas class has three pointer event handling methods (correspond to three pointer events' types):

protected void pointerPressed(int x, int y)

When the user touch a specific point of the screen.

protected void pointerReleased(int x, int y)

When the user release a specific touched point.

protected void pointerDragged(int x, int y)

When the user drags the touched point over the screen.

The Canvas class enables the midlet to check whether the MIDP implementation on which it runs support the pointer events by calling the following method:

public boolean hasPointerEvents()

The following midlet presents a simple pointer event handling. Try it on your palm...

```
//filename:PointerDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;
```

```
import javax.microedition.io.*;
import java.util.*;
```

```
public class PointerDemo extends MIDlet implements CommandListener
{
    private Display display;
    private Form form;
    private Command exitCommand, backCommand, startCommand;
    private PointerCanvas pointerCanvas;

    public PointerDemo()
    {
        //create the commands
        exitCommand = new Command("Exit", Command.EXIT, 1);
        backCommand = new Command("Back", Command.BACK, 1);
        startCommand = new Command("Start", Command.OK, 1);

        //creating the canvas
        pointerCanvas = new PointerCanvas();
        pointerCanvas.addCommand(backCommand);

        //creating the form
        form = new Form("KEY EVENTS");
        form.addCommand(exitCommand);
        form.addCommand(startCommand);
        System.out.println("commands were added");

        //setting the listeners
        form.setCommandListener(this);
        pointerCanvas.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void commandAction(Command c, Displayable d)
    {
        System.out.println("within commandAction");
        if(c==exitCommand)
        {
            System.out.println("exitCommand was pressed");
            exit();
        }
        else
        {

```

```

        if(c==startCommand)
        {
            System.out.println("startCommand was pressed");
            display.setCurrent(pointerCanvas);
        }
        else
        {
            if(c==backCommand)
            {
                display.setCurrent(form);
            }
        }
    }
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void startApp()
{
    display.setCurrent(form);
    System.out.println("form was set as the current screen");
}

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    display = null;
    form = null;
    exitCommand = null;
    backCommand = null;
    startCommand = null;
    pointerCanvas = null;
}
}

class PointerCanvas extends Canvas
{
    private int width;

```

```
private int height;
private Random random;
private int fontHeight;
private Image img;
private int imgWidth;
private int imgHeight;
private int x,y;
```

```
PointerCanvas()
{
    width = this.getWidth();
    try
    {
        img = Image.createImage("/bird.png");
    }
    catch(Exception exception)
    {
        exception.printStackTrace();
    }
    width = getWidth();
    height = getHeight();
    x = width/2;
    y = height/2;
    imgHeight = img.getHeight()+2;
    imgWidth = img.getWidth()+2;
}
```

```
protected void pointerPressed(int x, int y)
{
    this.x = x;
    this.y = y;
    repaint();
}
```

```
protected void keyPressed(int keyCode)
{
    //handling the game action
    System.out.println("within the keyPressed");
    switch(getGameAction(keyCode))
    {
        case Canvas.DOWN:
            y++;
            break;
        case Canvas.UP:
            y--;
            break;
    }
}
```

```

        case Canvas.LEFT:
            x--;
            break;
        case Canvas.RIGHT:
            x++;
            break;
    }
    repaint(x-imgWidth/2,y-imgHeight/2,imgWidth,imgHeight);
}

public void paint(Graphics g)
{
    System.out.println("within the paint method");
    g.setColor(0);
    g.fillRect(0,0,width,height);
    g.drawImage(img, x, y, Graphics.VCENTER|Graphics.HCENTER);
}
}

```

Animation Techniques and Drawing

Buffering

When changes are made to the canvas the midlet has to call one of the repaint methods in order to make the paint method paint the canvas surface. The repaint method returns without waiting for the paint method to finish (the invocation of the paint() method by the repaint() is done asynchronously).

The way of calling the paint() method synchronously is calling one of the following methods:

```
public void callSerially(Runnable r)
```


Calling this method, which was declared in the Display class, cause r to run immediately after the paint method is returned. Calling this method makes sure that right after the paint method is returned the method run() will be invoked on r.

public final void serviceRepaint()

Calling this method forces any repaint request to be serviced immediately. Calling this method blocks the thread in which it happens until all the pending repaint requests are served and the repaint is finished or if this canvas is not visible. If this canvas is not visible then calling this method does nothing and returns immediately. When using this method be care of deadlocks situations. After all, if the current thread holds a lock that the paint method need then a dead lock will happens.

The following midlet presents a simple use of the callSerially() method in order to generate animation smoothly as possible.

```
//filename:AnimationDemo.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;  
import javax.microedition.io.*;
```

```
public class AnimationDemo extends MIDlet implements CommandListener  
{  
    private Display display;  
    private Form form;  
    private Command exitCommand, backCommand, startCommand;  
    private AnimationCanvas animationCanvas;
```

```

public AnimationDemo()
{
    //create the commands
    exitCommand = new Command("Exit", Command.EXIT, 1);
    backCommand = new Command("Back", Command.BACK, 1);
    startCommand = new Command("Start", Command.OK, 1);

    //creating the canvas
    animationCanvas = new AnimationCanvas();
    animationCanvas.addCommand(backCommand);
    System.out.println("The image canvas was created");

    //creating the form
    form = new Form("BIRDY");
    form.addCommand(exitCommand);
    form.addCommand(startCommand);
    System.out.println("commands were added");

    //setting the listeners
    form.setCommandListener(this);
    animationCanvas.setCommandListener(this);

    //getting the display object
    display = Display.getDisplay(this);
}

public void commandAction(Command c, Displayable d)
{
    System.out.println("within commandAction");
    if(c==exitCommand)
    {
        System.out.println("exitCommand was pressed");
        exit();
    }
    else
    {
        if(c==startCommand)
        {
            System.out.println("startCommand was pressed");
            display.setCurrent(animationCanvas);
            animationCanvas.startAnimation();
        }
        else
        {
            if(c==backCommand)
            {

```

```

        display.setCurrent(form);
        animationCanvas.stopAnimation();
    }
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void startApp()
{
    display.setCurrent(form);
    System.out.println("form was set as the current screen");
}

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    display = null;
    form = null;
    exitCommand = null;
    backCommand = null;
    startCommand = null;
    animationCanvas = null;
}

class AnimationCanvas extends Canvas implements Runnable
{
    private int width;
    private int height;
    private Image img[];
    private int imgCounter;
    private boolean goOn;

    AnimationCanvas()
    {
        imgCounter = 0;
        width = this.getWidth();
        height = this.getHeight();
    }
}

```

```

        img = new Image[8];
        try
        {
            for(int i=0; i<8; i++)
            {
                img[i] = Image.createImage("/bird"+(i+1)+".png");
                System.out.println("created image:
bird"+(i+1)+".png");
            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }

    void startAnimation()
    {
        goOn = true;
        repaint();
        display.callSerially(this);
    }

    void stopAnimation()
    {
        goOn = false;
    }

    public void run()
    {
        try
        {
            Thread.sleep(100);
            imgCounter++;
            if(imgCounter>=img.length)
            {
                imgCounter=0;
            }
            repaint();
            if(goOn)
            {
                display.callSerially(this);
            }
        }
        catch(Exception exception)
        {

```

```

        exception.printStackTrace();
    }
}

public void paint(Graphics graphics)
{
    graphics.drawImage(img[imgCounter],width/2,height/2,
        graphics.HCENTER|graphics.VCENTER);
}
}
}

```

Calling the `serviceRepaints()` method flush any pending repaint requests.

Public final void serviceRepaints()

As a result of calling this method, several repaint requests might cause one single invocation of `paint()`. The `serviceRepaints()` method blocks until all the pending requests have been serviced and the `paint()` method returns. Severe result of this way of action is several repaint requests that result in one `paint()` invocation and cause, by that, a corrupted screen.

Solution to this problem might be using the synchronization mechanism together with calling the `serviceRepaints()` method. Using the synchronization mechanism it is possible to guarantee that no other thread will call the `repaint()` method as long as the current call has not ended. This can simply achieved be by calling the repaint method from within a synchronization block that locks itself on the same object, that every other synchronization block - around other calls to `repaint()` - lock it self. Furthermore, placing a call to the `serviceRepaints()` method right after the call to `repaint` will guarantee that the paint method returns before the synchronization block ends. The following example presents this technique.

//filename:SyncAnimationDemo.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.

//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;
import java.util.*;
```

```
public class SyncAnimationDemo extends MIDlet implements CommandListener
{
    private Display display;
    private Form form;
    private Command exitCommand, backCommand, startCommand;
    private AnimationCanvas animationCanvas;

    public SyncAnimationDemo()
    {
        //create the commands
        exitCommand = new Command("Exit", Command.EXIT, 1);
        backCommand = new Command("Back", Command.BACK, 1);
        startCommand = new Command("Start", Command.OK, 1);

        //creating the canvas
        animationCanvas = new AnimationCanvas();
        animationCanvas.addCommand(backCommand);
        //System.out.println("The image canvas was created");

        //creating the form
        form = new Form("BIRDY");
        form.addCommand(exitCommand);
        form.addCommand(startCommand);
        //System.out.println("commands were added");

        //setting the listeners
        form.setCommandListener(this);
        animationCanvas.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void commandAction(Command c, Displayable d)
```

```

{
    //System.out.println("within commandAction");
    if(c==exitCommand)
    {
        //System.out.println("exitCommand was pressed");
        exit();
    }
    else
    {
        if(c==startCommand)
        {
            //System.out.println("startCommand was pressed");
            display.setCurrent(animationCanvas);
            animationCanvas.startAnimation();
        }
        else
        {
            if(c==backCommand)
            {
                display.setCurrent(form);
                animationCanvas.stopAnimation();
            }
        }
    }
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void startApp()
{
    display.setCurrent(form);
    //System.out.println("form was set as the current screen");
}

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    display = null;
    form = null;
}

```

```

        exitCommand = null;
        backCommand = null;
        startCommand = null;
        animationCanvas = null;
    }

```

```

class AnimationCanvas extends Canvas implements Runnable
{
    private int width;
    private int height;
    private Image img[];
    private int imgCounter;
    private boolean goOn;
    private int x,y;
    private BirdPositionMovement bpm;
    private Thread thread;

    AnimationCanvas()
    {
        bpm = new BirdPositionMovement();
        imgCounter = 0;
        width = this.getWidth();
        height = this.getHeight();
        x = width/2;
        y = height/2;
        img = new Image[8];
        try
        {
            for(int i=0; i<8; i++)
            {
                img[i] = Image.createImage("/bird"+(i+1)+".png");
                //System.out.println("created image:
bird"+(i+1)+".png");
            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }

    void startAnimation()
    {
        if(thread==null)
        {
            thread = new Thread(this);

```



```

        thread.start();
    }
    bpm.startPositionMovement();
    goOn = true;
}

void stopAnimation()
{
    if(thread!=null)
    {
        thread = null;
    }
    bpm.stopPositionMovement();
    goOn = false;
}

public void run()
{
    while(goOn)
    {
        try
        {
            Thread.sleep(100);
            imgCounter++;
            if(imgCounter>=img.length)
            {
                imgCounter=0;
            }
            //System.out.println("imgCounter="+imgCounter);
            synchronized(this)
            {
                repaint();
                serviceRepaints();
            }
        }
        catch(Exception exception)
        {
            exception.printStackTrace();
        }
    }
}

public void paint(Graphics graphics)
{

```

```

//System.out.println("Thread.currentThread()"+Thread.currentThread());
        graphics.drawImage(img[imgCounter],x,y,
            graphics.HCENTER|graphics.VCENTER);
    }

    class BirdPositionMovement extends Thread
    {
        private boolean cont = true;
        private Random random;

        BirdPositionMovement()
        {
            random = new Random();
        }

        void startPositionMovement()
        {
            cont = true;
            start();
        }

        void stopPositionMovement()
        {
            cont = false;
        }

        public void run()
        {
            while(cont)
            {
                try
                {
                    Thread.sleep(100);
                }
                catch(Exception e)
                {
                    e.printStackTrace();
                }
                synchronized(AnimationCanvas.this)
                {
                    int xChange = random.nextInt()%2;
                    int yChange = random.nextInt()%2;
                    x += xChange;
                    y -= yChange;
                }
            }
        }
    }

```

```
//System.out.println("BirdPositionMovement:
xChange="+xChange+"yChange="+yChange);
AnimationCanvas.this.repaint();

AnimationCanvas.this.serviceRepaints();
}
}
}
}
}
```

Sometimes, the execution speed is not satisfied (rather slow) and the screen shows some nervous flickering signs. This flickering can be eliminated using the "double buffering" technique. Double buffering means that instead of rendering directly to the drawing area of the canvas, an off screen image is created and only when it is finished the midlet copies the created off screen image to the canvas. This way, each time the buffer is rendered to the real screen it is faster than doing it directly to the screen without using a buffer.

Some of the MIDP implementation already uses the double buffering technique. Using the `isDoubleBuffered()` method (declared in `Graphics`) you can check whether the double buffering technique already exist in the given MIDP implementation. In case the double buffering technique doesn't exist it is very simple to implement it. The following example presents a simple midlet that implements the double buffering technique. Note that the double buffering technique consumes a lot of memory and CPU power.

```
//filename:DoubleBuffering.java
```

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any form or by any means without the

```
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;
import java.util.*;
```

```
public class DoubleBuffering extends MIDlet implements CommandListener
{
    private Display display;
    private Form form;
    private Command exitCommand, backCommand, startCommand;
    private TableCanvas tableCanvas;

    public DoubleBuffering()
    {
        //create the commands
        exitCommand = new Command("Exit", Command.EXIT, 1);
        backCommand = new Command("Back", Command.BACK, 1);
        startCommand = new Command("Start", Command.OK, 1);

        //creating the canvas
        tableCanvas = new TableCanvas();
        tableCanvas.addCommand(backCommand);
        //System.out.println("The image canvas was created");

        //creating the form
        form = new Form("TABLE");
        form.addCommand(exitCommand);
        form.addCommand(startCommand);
        //System.out.println("commands were added");

        //setting the listeners
        form.setCommandListener(this);
        tableCanvas.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void commandAction(Command c, Displayable d)
    {
        //System.out.println("within commandAction");
    }
}
```

```

        if(c==exitCommand)
        {
            //System.out.println("exitCommand was pressed");
            exit();
        }
        else
        {
            if(c==startCommand)
            {
                //System.out.println("startCommand was pressed");
                display.setCurrent(tableCanvas);
            }
            else
            {
                if(c==backCommand)
                {
                    display.setCurrent(form);
                }
            }
        }
    }

    public void exit()
    {
        destroyApp(true);
        notifyDestroyed();
    }

    public void startApp()
    {
        display.setCurrent(form);
        //System.out.println("form was set as the current screen");
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        form = null;
        exitCommand = null;
        backCommand = null;
        startCommand = null;
        tableCanvas = null;
    }

```

```

    }

class TableCanvas extends Canvas
{
    private int width;
    private int height;
    private Graphics graphicsImage;
    private Image img;

    TableCanvas()
    {
        width = this.getWidth();
        height = this.getHeight();
        img = Image.createImage(width,height);
        graphicsImage = img.getGraphics();
    }

    public void paint(Graphics g)
    {
        graphicsImage.setColor(0xFFFFFFFF);
        graphicsImage.fillRect(0,0,width,height);

        int stepX = width / 10;
        int stepY = height / 10;

        graphicsImage.setColor(0);
        graphicsImage.drawRect(0,0,width,height);
        for(int y=0; y<height; y+=stepY)
        {
            graphicsImage.drawLine(0,y,width,y);
        }
        for(int x=0; x<width; x+=stepX)
        {
            graphicsImage.drawLine(x,0,x,height);
        }
        g.drawImage(img,0,0,g.TOP|g.LEFT);
    }
}
}

```

Chapter 7:

Performance

- [Introduction](#)
- [Benchmarking Tests](#)
- [Strategies and Techniques](#)

Introduction

Unlike stand-alone application, applets, servlets, EJB's etc... the midlet runs on a smaller platform (less memory and less CPU power). For that reason, when developing midlets it is extremely important making them fast and lean. Therefore, this chapter is devoted to performance issues. Lots of the techniques that relevant to developing on the J2SE and on the J2EE are relevant to the J2ME as well. However, some of these techniques deserve special attention when dealing with the J2ME. As was said, the available memory and CPU power on the MIDP implementation are rather small.

Benchmarking Tests

As of now, as I write these lines, there are no benchmark tools for testing midlets. Therefore, the benchmarking test that you can do is simple tests using the following methods in the Runtime and in the System classes:

In Runtime:

public long freeMemory()

This method returns an approximation to the total amount of free memory (for holding objects) the system has (measured in bytes).

public long totalMemory()

This method returns the total amount of memory (for holding objects) the system has (measured in bytes).

In System:

public static long currentTimeMillis()

This method returns the current time measured in milliseconds that have passed since January 1, 1970.

Using these two methods you can check the memory usage and the speed of your midlet. The following midlet presents a simple use of these two methods.

//filename: BenchmarkingTest.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any firstForm or by any means without the
//written permission of the publisher.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;
import java.util.*;
```

```
public class BenchmarkingTest extends MIDlet implements CommandListener
{
    private Display display;
    private Form firstForm, secondForm;
    private Command exitCommand, backCommand, startCommand;
    private TableCanvas tableCanvas;

    public BenchmarkingTest()
    {
        //create the commands
        exitCommand = new Command("Exit", Command.EXIT, 1);
        backCommand = new Command("Back", Command.BACK, 1);
        startCommand = new Command("Start", Command.OK, 1);
    }
}
```

```

        //creating the firstForm
        firstForm = new Form("BENCHMARKING");
        firstForm.addCommand(exitCommand);
        firstForm.addCommand(startCommand);

        //creating the secondForm
        secondForm = new Form("RESULTS");
        secondForm.addCommand(backCommand);

        //setting the listeners
        firstForm.setCommandListener(this);
        secondForm.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void commandAction(Command c, Displayable d)
    {
        //System.out.println("within commandAction");
        if(c==exitCommand)
        {
            //System.out.println("exitCommand was pressed");
            exit();
        }
        else
        {
            if(c==startCommand)
            {
                Runtime runtime = Runtime.getRuntime();
                long memBefore, memAfter, totalMem, startTime,
endTime;

                totalMem = runtime.totalMemory();
                memBefore = runtime.freeMemory();
                startTime = System.currentTimeMillis();
                tableCanvas = new TableCanvas();
                endTime = System.currentTimeMillis();
                memAfter = runtime.freeMemory();
                secondForm.append(new StringItem("Total
Memory",String.valueOf(totalMem)));
                secondForm.append(new StringItem("Free
Memory",String.valueOf(memAfter)));
                secondForm.append(new StringItem("Memory Used by the
TabalCanvas Object",

```

```
String.valueOf(memBefore-memAfter)));
        secondForm.append(new StringItem("Time it took to
instantiate the TableCanvas object",
```

```
String.valueOf(endTime-startTime)));
        display.setCurrent(secondForm);
    }
    else
    {
        if(c==backCommand)
        {
            display.setCurrent(firstForm);
        }
    }
}
```

```
public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}
```

```
public void startApp()
{
    display.setCurrent(firstForm);
    //System.out.println("firstForm was set as the current screen");
}
```

```
public void pauseApp()
{
}
```

```
public void destroyApp(boolean cond)
{
    display = null;
    firstForm = null;
    exitCommand = null;
    backCommand = null;
    startCommand = null;
    tableCanvas = null;
}
```

```
class TableCanvas extends Canvas
{
```

```

private int width;
private int height;
private Graphics graphicsImage;
private Image img;

TableCanvas()
{
    width = this.getWidth();
    height = this.getHeight();
    img = Image.createImage(width,height);
    graphicsImage = img.getGraphics();
    graphicsImage.setColor(0xFFFFFFFF);
    graphicsImage.fillRect(0,0,width,height);

    int stepX = width / 10;
    int stepY = height / 10;

    graphicsImage.setColor(0);
    graphicsImage.drawRect(0,0,width,height);
    for(int y=0; y<height; y+=stepY)
    {
        graphicsImage.drawLine(0,y,width,y);
    }
    for(int x=0; x<width; x+=stepX)
    {
        graphicsImage.drawLine(x,0,x,height);
    }
}

public void paint(Graphics g)
{
    g.drawImage(img,0,0,g.TOP|g.LEFT);
}
}

```

Strategies and Techniques

In this section I will cover several strategies and techniques that are relevant to improving the midlet execution.

Using the StringBuffer

Each time two strings are concatenating using the plus operator, usually behind the scenes, new String and StringBuffer objects are created. Because of that, special attention should be given to those cases in which new String objects are created. Sometimes the usage of a StringBuffer might improve the performance. The StringBuffer object represents a string that can be changed using several different methods (unlike the String object that represents an unchangeable string). The following midlet present the creation of a new String object that holds the "abcd...z" string. The midlet enables the user to choose whether to use the StringBuffer class in creating that string. Try it and see the improvements you get when using a string buffer (instead of doing a simple string concatenating).

```
//filename:SringBufferUsage.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any firstForm or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;  
import javax.microedition.io.*;  
import java.util.*;
```

```
public class SringBufferUsage extends MIDlet implements CommandListener  
{  
    private Display display;  
    private Form firstForm, secondForm;  
    private Command exitCommand, backCommand, withStringBufferCommand,  
withoutStringBufferCommand;
```

```

public StringBufferUsage()
{
    //create the commands
    exitCommand = new Command("Exit", Command.EXIT, 1);
    backCommand = new Command("Back", Command.BACK, 1);
    withStringBufferCommand = new Command("With String Buffer",
Command.SCREEN, 1);
    withoutStringBufferCommand = new Command("Without String Buffer",
Command.SCREEN, 1);

    //creating the firstForm
    firstForm = new Form("StringBufferUsage");
    firstForm.addCommand(exitCommand);
    firstForm.addCommand(withStringBufferCommand);
    firstForm.addCommand(withoutStringBufferCommand);

    //creating the secondForm
    secondForm = new Form("Results");
    secondForm.addCommand(backCommand);

    //setting the listeners
    firstForm.setCommandListener(this);
    secondForm.setCommandListener(this);

    //getting the display object
    display = Display.getDisplay(this);
}

public void commandAction(Command c, Displayable d)
{
    //System.out.println("within commandAction");
    if(c==exitCommand)
    {
        //System.out.println("exitCommand was pressed");
        exit();
    }
    else
    {
        if(c==withoutStringBufferCommand)
        {
            Runtime runtime;
            long startTime, endTime;
            if(secondForm.size()==1)
            {
                secondForm.delete(0);
            }
        }
    }
}

```

```

    }
    runtime = Runtime.getRuntime();
    startTime = System.currentTimeMillis();
    //creating the "abcd...z" string
    String result = new String("");
    for(int i=0; i<100; i++)
    {
        for(char ch='a'; ch<'z'; ch++)
        {
            result += ch;
        }
    }
    System.out.println(result);
    endTime = System.currentTimeMillis();
    secondForm.append(new StringItem("Time in millis : ",
String.valueOf(endTime-startTime)));
    display.setCurrent(secondForm);
}
else
{
    if(c==withStringBufferCommand)
    {
        Runtime runtime;
        long startTime, endTime;
        if(secondForm.size()==1)
        {
            secondForm.delete(0);
        }
        runtime = Runtime.getRuntime();
        startTime = System.currentTimeMillis();
        System.out.println("startTime="+startTime);
        //creating the "abcd...z" string
        StringBuffer sb = new StringBuffer("");
        for(int i=0; i<100; i++)
        {
            for(char ch='a'; ch<'z'; ch++)
            {
                sb.append(ch);
            }
        }
        String result = sb.toString();
        System.out.println(result);
        endTime = System.currentTimeMillis();
        System.out.println("endTime="+endTime);
        secondForm.append(new StringItem("Time in
millis : ", String.valueOf(endTime-startTime)));
    }
}

```

```

        display.setCurrent(secondForm);
    }
    else
    {
        if(c==backCommand)
        {
            display.setCurrent(firstForm);
        }
    }
}

}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void startApp()
{
    display.setCurrent(firstForm);
    //System.out.println("firstForm was set as the current screen");
}

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    display = null;
    firstForm = null;
    secondForm = null;
    exitCommand = null;
    backCommand = null;
    withStringBufferCommand = null;
    withoutStringBufferCommand = null;
}
}

```

Avoiding unnecessary objects

Each time a new object is created, memory is allocated and this allocation takes time. Furthermore, when there is no reference to a given object, the garbage collector running also takes time. Therefore, special attention should be given to unnecessary objects creation. The following midlet presents two scenarios. In the first scenario, the Rectangle class is instantiated within a loop. For that, new object is created each iteration and reallocated as the scope of the variable that holds its reference ends. In the second scenario, the Rectangle class is instantiated only once (before the loop starts). The two scenarios do the same. The midlet presents the time in millis that took each of these scenarios to run.

```
//filename:TooManyObjects.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any form or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;  
import javax.microedition.io.*;  
import java.util.*;
```

```
public class TooManyObjects extends MIDlet implements CommandListener  
{  
    private Display display;  
    private Form firstForm, secondForm;  
    private Command exitCommand, backCommand, objectWithinLoop,  
noObjectWithinLoop;  
  
    public TooManyObjects()  
    {  
        //create the commands  
        exitCommand = new Command("Exit", Command.EXIT, 1);  
        backCommand = new Command("Back", Command.BACK, 1);  
        objectWithinLoop = new Command("objects within loop",  
Command.SCREEN, 1);
```

```
        noObjectWithinLoop = new Command("no objects within loop",  
Command.SCREEN, 1);
```

```
        //creating the firstForm  
        firstForm = new Form("RedundantObjects");  
        firstForm.addCommand(exitCommand);  
        firstForm.addCommand(objectWithinLoop);  
        firstForm.addCommand(noObjectWithinLoop);
```

```
        //creating the secondForm  
        secondForm = new Form("Results");  
        secondForm.addCommand(backCommand);
```

```
        //setting the listeners  
        firstForm.setCommandListener(this);  
        secondForm.setCommandListener(this);
```

```
        //getting the display object  
        display = Display.getDisplay(this);
```

```
    }
```

```
    public void commandAction(Command c, Displayable d)  
    {
```

```
        //System.out.println("within commandAction");  
        if(c==exitCommand)  
        {  
            //System.out.println("exitCommand was pressed");  
            exit();  
        }
```

```
        else
```

```
        {  
            if(c==noObjectWithinLoop)  
            {  
                Runtime runtime;  
                long startTime, endTime;  
                MyRectangle rec;  
                long sum = 0;  
                if(secondForm.size()==1)  
                {  
                    secondForm.delete(0);  
                }  
                runtime = Runtime.getRuntime();  
                startTime = System.currentTimeMillis();  
  
                rec = new MyRectangle();  
                for(int i=1; i<=50; i++)
```

```

        {
            for(int j=1; j<=55; j++)
            {
                rec.setWidth(i);
                rec.setHeight(j);
                sum    += rec.area();
            }
        }
        System.out.println("sum="+sum);

        endTime = System.currentTimeMillis();
        secondForm.append(new StringItem("Time in millis : ",
String.valueOf(endTime-startTime)));
        System.out.println("sum="+sum+ "
time="+String.valueOf(endTime-startTime));
        display.setCurrent(secondForm);
    }
    else
    {
        if(c==objectWithinLoop)
        {
            Runtime runtime;
            long startTime=0, endTime=0;
            MyRectangle rec = new MyRectangle();
            long sum = 0;
            if(secondForm.size()==1)
            {
                secondForm.delete(0);
            }
            startTime = System.currentTimeMillis();
            for(int i=1; i<=50; i++)
            {
                for(int j=1; j<=55; j++)
                {
                    rec = new MyRectangle();
                    rec.setWidth(i);
                    rec.setHeight(j);
                    sum    += rec.area();
                }
            }
            System.out.println("sum="+sum);

            endTime = System.currentTimeMillis();
            secondForm.append(new StringItem("Time in
millis : ", String.valueOf(endTime-startTime)));

```

```

        System.out.println("sum="+sum+ "
time="+String.valueOf(endTime-startTime));
        display.setCurrent(secondForm);
    }
    else
    {
        if(c==backCommand)
        {
            display.setCurrent(firstForm);
        }
    }
}

}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void startApp()
{
    display.setCurrent(firstForm);
    //System.out.println("firstForm was set as the current screen");
}

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    display = null;
    firstForm = null;
    secondForm = null;
    exitCommand = null;
    backCommand = null;
    objectWithinLoop = null;
    noObjectWithinLoop = null;
}
}

```

```

class MyRectangle
{

```

```

private int width=0;
private int height=0;

MyRectangle()
{
}

MyRectangle(int width, int height)
{
    this.width = width;
    this.height = height;
}

void setWidth(int w)
{
    width = w;
}

void setHeight(int h)
{
    height = h;
}

int area()
{
    return width*height;
}
}

```

Redundant method calls

Each time the midlet calls a method it takes time. Sometimes, the code includes redundant method calls. Identifying these redundant method calls and fix them optimize the code. The following midlet presents a redundant method call. The midlet presents the same sorting algorithm in two versions. In the first, the size() method is called within a loop. In the second, the size() method isn't called within the loop and yet it does the same.

//filename:RedundantMethodCalls.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any firstForm or by any means without the
//written permission of the publisher.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;
import java.util.*;
```

```
public class RedundantMethodCalls extends MIDlet implements CommandListener
{
    private Display display;
    private Form firstForm, secondForm;
    private Command exitCommand, backCommand,
    redundantMehodCallsCommand,
        notRedundantMehodCallsCommand;
    private MyRectangle recs[];

    public RedundantMethodCalls()
    {
        //create the commands
        exitCommand = new Command("Exit", Command.EXIT, 1);
        redundantMehodCallsCommand = new Command("Redundant",
Command.OK, 1);
        backCommand = new Command("Back", Command.BACK, 1);
        notRedundantMehodCallsCommand = new Command("Not Redundant",
Command.OK, 1);
        firstForm = new Form("RedundantObjects");
        firstForm.addCommand(exitCommand);
        firstForm.addCommand(redundantMehodCallsCommand);
        firstForm.addCommand(notRedundantMehodCallsCommand);

        //creating the secondForm
        secondForm = new Form("Results");
        secondForm.addCommand(backCommand);

        //setting the listeners
        firstForm.setCommandListener(this);
        secondForm.setCommandListener(this);
    }
}
```



```

        vec.setElementAt(vec.elementAt(j),i);
                                vec.setElementAt(temp,j);
                                }
                                }
                                }
                                long end = System.currentTimeMillis();
                                secondForm.append(new
StringItem("time=",String.valueOf(end-start)));
                                display.setCurrent(secondForm);
                                }//0
                                else
                                {
                                if(c==notRedundantMehodCallsCommand)
                                {

                                System.out.println("notRedundantMehodCallsCommand");
                                long start = System.currentTimeMillis();
                                int size = vec.size();
                                for(int i=0; i<size; i++)
                                {
                                for(int j=i+1; j<size; j++)
                                {

                                if((((MyRectangle)vec.elementAt(j)).area())>

                                ((MyRectangle)vec.elementAt(i)).area())

                                {
                                Object temp =
vec.elementAt(i);

                                vec.setElementAt(vec.elementAt(j),i);

                                vec.setElementAt(temp,j);

                                }

                                }

                                }
                                long end = System.currentTimeMillis();
                                secondForm.append(new
StringItem("time=",String.valueOf(end-start)));
                                display.setCurrent(secondForm);
                                }
                                }//0
                                }//1
                                }//2
                                }

```



```

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void startApp()
{
    display.setCurrent(firstForm);
    //System.out.println("firstForm was set as the current screen");
}

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    display = null;
    firstForm = null;
    secondForm = null;
    exitCommand = null;
    notRedundantMehodCallsCommand = null;
    redundantMehodCallsCommand = null;
    backCommand = null;
}
}

```

```

class MyRectangle
{
    private int width=0;
    private int height=0;

    MyRectangle()
    {
    }

    MyRectangle(int width, int height)
    {
        this.width = width;
        this.height = height;
    }

    void setWidth(int w)

```

```

    {
        width = w;
    }

    void setHeight(int h)
    {
        height = h;
    }

    int area()
    {
        return width*height;
    }
}

```

Usage of simple Arrays

When the number of items is fixed and known, using simple arrays is more efficient than using one of the Collection classes. The following midlet presents that.

//filename:ArraysUsing.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
 //All rights reserved. No part of the contents of this program may be
 //reproduced or transmitted in any firstForm or by any means without the
 //written permission of the publisher.

```

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;
import java.util.*;

```

```

public class ArraysUsing extends MIDlet implements CommandListener
{
    private Display display;
    private Form firstForm, secondForm;
    private Command exitCommand, backCommand, usingArrayCommand,
        usingVectorCommand;

```

```

private MyRectangle recs[];

public ArraysUsing()
{
    //create the commands
    exitCommand = new Command("Exit", Command.EXIT, 1);
    usingArrayCommand = new Command("usingArray", Command.OK, 1);
    backCommand = new Command("Back", Command.BACK, 1);
    usingVectorCommand = new Command("usingVector", Command.OK,
1);

    firstForm = new Form("RedundantObjects");
    firstForm.addCommand(exitCommand);
    firstForm.addCommand(usingArrayCommand);
    firstForm.addCommand(usingVectorCommand);

    //creating the secondForm
    secondForm = new Form("Results");
    secondForm.addCommand(backCommand);

    //setting the listeners
    firstForm.setCommandListener(this);
    secondForm.setCommandListener(this);

    //getting the display object
    display = Display.getDisplay(this);
}

public void commandAction(Command c, Displayable d)
{
    //System.out.println("within commandAction");
    if(c==exitCommand)
    {
        //System.out.println("exitCommand was pressed");
        exit();
    }
    else
    {
        //2
        if(c==backCommand)
        {
            System.out.println("backCommand");
            display.setCurrent(firstForm);
        }
        else
        {
            //1
            Random random = new Random();

```

```

        if(c==usingArrayCommand)
        {
            //0
            System.out.println("usingArrayCommand");
            if(secondForm.size()==1)
            {
                secondForm.delete(0);
            }
            MyRectangle arr[];
            arr = new MyRectangle[2000];
            for(int i=0; i<2000; i++)
            {
                arr[i] = new MyRectangle(
Math.abs(random.nextInt()%10),
Math.abs(random.nextInt()%10));
            }

            long start = System.currentTimeMillis();
            int siz = arr.length;
            for(int i=0; i<siz; i++)
            {
                for(int j=i+1; j<siz; j++)
                {
                    if(arr[j].area()>arr[i].area())
                    {
                        MyRectangle temp = arr[i];
                        arr[i] = arr[j];
                        arr[j] = temp;
                    }
                }
            }
            long end = System.currentTimeMillis();
            secondForm.append(new
StringItem("time=",String.valueOf(end-start)));
            display.setCurrent(secondForm);
        }
        //0
        else
        {
            //0
            if(c==usingVectorCommand)
            {
                System.out.println("usingVectorCommand");
                if(secondForm.size()==1)
                {
                    secondForm.delete(0);
                }
            }
        }
    }
}

```

```

Vector vec = new Vector();
for(int i=0; i<2000; i++)
{
    vec.addElement(new MyRectangle(
Math.abs(random.nextInt()%10),
Math.abs(random.nextInt()%10)));
}

long start = System.currentTimeMillis();
int size = vec.size();
for(int i=0; i<size; i++)
{
    for(int j=i+1; j<size; j++)
    {
        if(((MyRectangle)vec.elementAt(j)).area()>
((MyRectangle)vec.elementAt(i)).area())
        {
            Object temp =
vec.elementAt(i);

            vec.setElementAt(vec.elementAt(j),i);

            vec.setElementAt(temp,j);

        }
    }
}
long end = System.currentTimeMillis();
secondForm.append(new
StringItem("time=",String.valueOf(end-start)));
display.setCurrent(secondForm);
}
} //0
} //1
} //2
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void startApp()
```

```

    {
        display.setCurrent(firstForm);
        //System.out.println("firstForm was set as the current screen");
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        firstForm = null;
        secondForm = null;
        exitCommand = null;
        usingVectorCommand = null;
        usingArrayCommand = null;
        backCommand = null;
    }
}

```

```

class MyRectangle
{
    private int width=0;
    private int height=0;

    MyRectangle()
    {
    }

    MyRectangle(int width, int height)
    {
        this.width = width;
        this.height = height;
    }

    void setWidth(int w)
    {
        width = w;
    }

    void setHeight(int h)
    {
        height = h;
    }
}

```

```

    int area()
    {
        return width*height;
    }
}

```

If using one of the *Collection* classes is still suitable, most of them enable customizing their way of action. The *Vector* class, for instance, enables setting its initial size in its initialization.

I\O Buffering

Reading or writing bytes one at a time takes more time than writing them in groups (group of bytes instead of single bytes). In J2SE you could use the *Buffered* streams for that purpose. However, the J2ME(CLDC\MIDP) doesn't supply these buffer streams so the only way of getting the buffering functionality is writing the needed code for that. The following two midlets present a simple case of reading bytes with (and with out) a buffer.

```
//filename:BufferUse.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
```

```
public class BufferUse extends MIDlet
{
    private static final int BUFFER_SIZE = 100;
```

```

private Display display;
private Command exitCommand, stopCommand;
private TextBox textBox;

public BufferUse()
{
    System.out.println("BufferUse");
    long start,end;
    byte buffer[] = new byte[BUFFER_SIZE];
    InputStream is = null;
    try
    {
        Class theClass = Class.forName("BufferUse");
        is = theClass.getResourceAsStream("/details.txt");
        StringBuffer sb = new StringBuffer();

        start = System.currentTimeMillis();

        while(true)
        {
            synchronized(buffer)
            {
                int amount = is.read(buffer);
                if(amount== -1)
                {
                    break;
                }
                System.out.println(new String(buffer));
            }
        }
        String str = sb.toString();

        end = System.currentTimeMillis();

        System.out.println("time="+ (end-start));
    }
    catch(Exception e)
    {
    }
    finally
    {
        if(is!=null)
        {
            try
            {
                is.close();
            }
        }
    }
}

```



```

                                is = null;
                                }
                                catch(Exception e)
                                {
                                }
                            }
                    }
    }

    public void startApp()
    {
        display = Display.getDisplay(this);
        display.setCurrent(textBox);
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
    }
}

```

//filename:NoBufferUse.java

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
 //All rights reserved. No part of the contents of this program may be
 //reproduced or transmitted in any form or by any means without the
 //written permission of the publisher.

```

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;

```

```

public class NoBufferUse extends MIDlet
{
    private Display display;
    private Command exitCommand, stopCommand;
    private TextBox textBox;

```

```

public NoBufferUse()
{
    System.out.println("NoBufferUse");
    InputStream is = null;
    try
    {
        long start, end;
        Class theClass = Class.forName("NoBufferUse");
        is = theClass.getResourceAsStream("/details.txt");

        start = System.currentTimeMillis();

        int temp = is.read();
        while(temp!=-1)
        {
            System.out.print((char)temp);
            temp = is.read();
        }

        end = System.currentTimeMillis();

        System.out.println("time="+end-start);
    }
    catch(Exception e)
    {
    }
    finally
    {
        if(is!=null)
        {
            try
            {
                is.close();
                is = null;
            }
            catch(IOException e)
            {
            }
        }
    }
}

public void startApp()
{
    display = Display.getDisplay(this);
}

```

```

        display.setCurrent(textBox);
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
    }
}

```

Even though, it seems that using a buffer is always more efficient, sometimes, not using a buffer might be more efficient. You should check each case separately. The way you intend to use the data you got has a substantial influence on the performance.

Resources Releasing

Releasing resources as soon as no one needs them can substantially improve the midlet performance. Releasing resources the midlet doesn't need means not only placing 'null' in all of the variables that hold the specific resource reference but it also means considering calling the `System.gc()` method as well as some other specific methods that immediately release the resource (e.g. calling `close()` when dealing with streams and network connections).

The best place for putting the `close()` method calling is in the finally block, which its code gets executed no matter what happens.

Another good design principle is putting each of the `close()` calls that reside in the finally block within a different try&catch block. This way, no matter what happens, each of the `close()` calls will happen. Consider the following two code pieces. In the

first, if the first close() calling fails then the other close() calling doesn't take place. In the second, no matter what happens, both of the close() calling take place:

Example 1:

```
try
{
    ...

}
catch(Exception e)
{
    ...
}
finally
{
    try
    {
        if(is!=null)
        {
            is.close();
        }
        if(dis!=null)
        {
            dis.close();
        }
    }
    catch(Exception e) { }
}
```

Example 2:

```
try
{
    ...

}
catch(Exception e)
{
    ...
}
finally
```

```

{
    if(is!=null)
    {
        try
        {
            is.close();
        }
        catch(Exception e) { }
    }
    if(dis!=null)
    {
        try
        {
            dis.close();
        }
        catch(Exception e) { }
    }
}

```

Inform the user when he needs to wait

Adding a progress bar, gauge or another GUI component to indicate that the user needs to wait (for a network connection - for instance) doesn't speed up the midlet but at least improves its impression. Doing so - in those cases the user needs to wait - optimizes the perceived speed of the midlet and therefore might prevent hurting the users' satisfaction.

Not including unnecessary classes in the midlet suite

The size of the midlet suite (the one the user upload to his device (cellular telephone, palm pilot etc...)) should be minimized in order to improve the user satisfaction. This can be achieved by including within the midlet suite only the needed classes. Note that this guideline is especially relevant when using third party classes (such as XML parser's classes). Tacking out from third party packages

those unused classes can only improve the midlet suite uploading time and therefore the user satisfaction.

Avoiding large number of inner classes when handling events

Java programmers tend to declare more than one inner class for events handling. When dealing with the J2SE this is usually not a problem. However, when dealing with the J2ME this can substantially decrease the midlet performance. More classes (inner class is like any other class) means more linking during runtime, and this decrease the performance.

Avoiding using the Exceptions handling mechanism

The Exceptions Handling mechanism decreases the performance. Whenever it is possible consider avoiding the use of the Exceptions Handling mechanism.

Avoid Contiguous Instanceof and Casts

In Java, casting a type or performing a call to instanceof performs runtime checking (Java is different from C++).

Declaring the variables where they are needed

Even though, declaring the local variables at the start of the method is a common style (it is often nice for code maintenance because all the variables are located in to one place and that helps noticing\dealing with them) it might, sometimes, lead to unnecessary code being executed. The following two pieces of code demonstrate it. When the first code runs the Rectangle class is instantiated in any case. When the

second code runs the Rectangle class is instantiated only if there is a need in doing so.

Code 1:

```
public void doSomething()
{
    Rectangle rec = new Rectangle();
    if (...)
    {
        rec.set(23,33);
        .
        .
        .
    }
    else
    {
        return;
    }
}
```

Code 2:

```
public void doSomething()
{
    if (...)
    {
        Rectangle rec = new Rectangle();
        rec.set(23,33);
        .
        .
        .
    }
    else
    {
        return;
    }
}
```

Avoid unnecessary synchronization

Java has several powerful synchronization features. However, calling a synchronized method costs more than calling one that is not synchronized. Note this guideline when dealing with threads.

Local Variables are less expensive to use

Accessing an attribute is more expensive than accessing a local variable. While the exact location of the instance variable is determined during runtime, the exact location of the local variable in the stack is determined already during compilation. Because of that, the access to the instance variable during runtime isn't done directly but using some kind of a look up mechanism. Therefore, when having a loop that access an instance variable it might be faster copying the value of the instance variable into a local variable and access directly to the local variable (instead of accessing the instance variable).

The following midlet presents this technique:

```
//filename:LocalInstanceVariable.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any firstForm or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;  
import javax.microedition.io.*;  
import java.util.*;
```

```
public class LocalInstanceVariable extends MIDlet implements CommandListener  
{  
    private Display display;  
    private Form firstForm, secondForm;
```



```

private Command exitCommand, backCommand, uusingLocalCommand,
        usingInstanceCommand;
private MyRectangle recs[];

public LocalInstanceVariable()
{
    //create the commands
    exitCommand = new Command("Exit", Command.EXIT, 1);
    uusingLocalCommand = new Command("uusingLocal", Command.OK,
1);
    backCommand = new Command("Back", Command.BACK, 1);
    usingInstanceCommand = new Command("usingInstance",
Command.OK, 1);
    firstForm = new Form("late-binding");
    firstForm.addCommand(exitCommand);
    firstForm.addCommand(uusingLocalCommand);
    firstForm.addCommand(usingInstanceCommand);

    //creating the secondForm
    secondForm = new Form("Results");
    secondForm.addCommand(backCommand);

    //setting the listeners
    firstForm.setCommandListener(this);
    secondForm.setCommandListener(this);

    //getting the display object
    display = Display.getDisplay(this);
}

public void commandAction(Command c, Displayable d)
{
    //System.out.println("within commandAction");
    if(c==exitCommand)
    {
        //System.out.println("exitCommand was pressed");
        exit();
    }
    else
    {
        //2
        if(c==backCommand)
        {
            System.out.println("backCommand");
            display.setCurrent(firstForm);
        }
        else

```

```

    { //1
        Random random = new Random();

        if(c==uusingLocalCommand)
        { //0
            System.out.println("uusingLocalCommand");
            if(secondForm.size()==1)
            {
                secondForm.delete(0);
            }
            MyRectangle arr[];
            arr = new MyRectangle[15000];

            long start = System.currentTimeMillis();

            int count = 0;
            for(int i=0; i<15000; i++)
            {
                arr[i] = new MyRectangle(
Math.abs(random.nextInt()%10),
Math.abs(random.nextInt()%10));
                count++;
            }

            MyRectangle.numOfRecs = count;
            System.out.println("num of rectangles is " +
MyRectangle.numOfRecs);

            long end = System.currentTimeMillis();
            secondForm.append(new
StringItem("time=",String.valueOf(end-start)));
            display.setCurrent(secondForm);
        } //0
        else
        { //0
            if(c==usingInstanceCommand)
            {

                System.out.println("usingInstanceCommand");
                if(secondForm.size()==1)
                {
                    secondForm.delete(0);
                }
                MyRectangle arr[];
                arr = new MyRectangle[15000];

```

```

        long start = System.currentTimeMillis();

        for(int i=0; i<15000; i++)
        {
            arr[i] = new MyRectangle(
Math.abs(random.nextInt()%10),

            Math.abs(random.nextInt()%10));
            MyRectangle.numOfRecs++;
        }

        System.out.println("num of rectangles is " +
MyRectangle.numOfRecs);

        long end = System.currentTimeMillis();
        secondForm.append(new
StringItem("time=",String.valueOf(end-start)));
        display.setCurrent(secondForm);
    }
} //0
} //1
} //2
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void startApp()
{
    display.setCurrent(firstForm);
    //System.out.println("firstForm was set as the current screen");
}

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    display = null;
    firstForm = null;
    secondForm = null;
    exitCommand = null;
    usingInstanceCommand = null;
}

```

```

        uusingLocalCommand = null;
        backCommand = null;
    }
}

```

```

class MyRectangle
{
    private int width=0;
    private int height=0;
    static int numOfRecs;

    MyRectangle()
    {
    }

    MyRectangle(int width, int height)
    {
        this.width = width;
        this.height = height;
    }

    void setWidth(int w)
    {
        width = w;
    }

    void setHeight(int h)
    {
        height = h;
    }

    int area()
    {
        return width*height;
    }
}

```

Consider doing the Computation on the server side

Running computationally intensive tasks on the device might be a bad choice comparing to the alternative which is sending the data to the server, let it do the computation task and return its answer back to the device.

In each case, you should consider and find the right balance between what to do on the device and what to do on the server. You should take into consideration both the application you deal with and both the device functionality and connectivity.

Object Pooling

Instead of continually allocating object and abandoning them later on, you can develop kind of object pooling system in which, when a new object is requested instead of instantiating it, an object that was created before will be returned. In the same time, object that finishes its tasks will be returned to the object pooling and will be save for later reuse. Using this technique less objects are created, and the midlet performance improves.

Using of Non-Object types (instead of class types) when ever possible

Each object is allocated from the heap during runtime. Therefore, each object that is allocated impacts the midlet performance as well as the memory usage. Reducing the number of objects can be achieved by using non-object types instead.

Therefore, you should consider each case of using an object whether you can use non-object type instead. The following midlet presents this idea.

```
//filename:ScalarTypes.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any firstForm or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;
import java.util.*;
```

```
public class ScalarTypes extends MIDlet implements CommandListener
{
    private Display display;
    private Form firstForm, secondForm;
    private Command exitCommand, backCommand, usingScalarsCommand,
        usingObjectsCommand;
    private MyLine recs[];

    public ScalarTypes()
    {
        //create the commands
        exitCommand = new Command("Exit", Command.EXIT, 1);
        usingScalarsCommand = new Command("uusingScalars", Command.OK,
1);
        backCommand = new Command("Back", Command.BACK, 1);
        usingObjectsCommand = new Command("usingInstance", Command.OK,
1);

        firstForm = new Form("SCALAR TYPES");
        firstForm.addCommand(exitCommand);
        firstForm.addCommand(usingScalarsCommand);
        firstForm.addCommand(usingObjectsCommand);

        //creating the secondForm
        secondForm = new Form("Results");
        secondForm.addCommand(backCommand);

        //setting the listeners
        firstForm.setCommandListener(this);
        secondForm.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
    }

    public void commandAction(Command c, Displayable d)
    {
        //System.out.println("within commandAction");
    }
}
```

```

        if(c==exitCommand)
        {
            //System.out.println("exitCommand was pressed");
            exit();
        }
        else
        {
            if(c==backCommand)
            {
                System.out.println("backCommand");
                display.setCurrent(firstForm);
            }
            else
            {
                Random random = new Random();

                if(c==usingScalarsCommand)
                {
                    System.out.println("usingScalarsCommand");
                    if(secondForm.size()==1)
                    {
                        secondForm.delete(0);
                    }
                    MyLine arr[];
                    arr = new MyLine[15000];

                    long start = System.currentTimeMillis();

                    for(int i=0; i<15000; i++)
                    {
                        arr[i] = new MyLine(
Math.abs(random.nextInt()%100),
Math.abs(random.nextInt()%100),
Math.abs(random.nextInt()%100),
Math.abs(random.nextInt()%100)

                        );
                    }

                    long end = System.currentTimeMillis();
                    secondForm.append(new
StringItem("time=",String.valueOf(end-start)));
                    display.setCurrent(secondForm);
                }
            }
        }
    }
}

```

```

        }//0
        else
        {
            if(c==usingObjectsCommand)
            {
                System.out.println("usingObjectsCommand");
                if(secondForm.size()==1)
                {
                    secondForm.delete(0);
                }
                MyLine arr[];
                arr = new MyLine[15000];

                long start = System.currentTimeMillis();

                for(int i=0; i<15000; i++)
                {
                    arr[i] = new MyLine(new
MyPoint(Math.abs(random.nextInt()%10),
            Math.abs(random.nextInt()%10)),
            new MyPoint(Math.abs(random.nextInt()%10),
            Math.abs(random.nextInt()%10)));
                }

                long end = System.currentTimeMillis();
                secondForm.append(new
StringItem("time=",String.valueOf(end-start)));
                display.setCurrent(secondForm);
            }
        }//0
    }//1
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void startApp()
{

```



```

        display.setCurrent(firstForm);
        //System.out.println("firstForm was set as the current screen");
    }

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        firstForm = null;
        secondForm = null;
        exitCommand = null;
        usingObjectsCommand = null;
        usingScalarsCommand = null;
        backCommand = null;
    }
}

```

```

class MyLine
{
    private int x1,y1;
    private int x2,y2;

    MyLine()
    {
    }

    MyLine(int x1, int y1, int x2, int y2)
    {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }

    MyLine(MyPoint p1, MyPoint p2)
    {
        x1 = p1.x;
        y1 = p1.y;
        x2 = p1.x;
        y2 = p1.y;
    }
}

```

```
class MyPoint
{
    int x,y;

    MyPoint(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

}
```

Chapter 8:

XML

- • [Introduction](#)
- • [TinyXML Parser for J2ME\(CLDC\MIDP\)](#)
- • [NanoXML Parser for J2ME\(CLDC\MIDP\)](#)

Introduction

XML is a markup language, similar to HTML only with one big difference (among other differences): in XML you can invent the TAGS so you are not restricted to group of TAGS that was already chosen and declared by others, as in HTML. These new TAGS you decide on when working with XML provides a great technique for creating textual documents that describe data. Java & XML is a great combination. Java makes the application portable among different platforms and XML makes the data portable among different applications.

Using XML in a wireless application (midlet) can dramatically reduce development cost as well as the efforts toward more interoperability and flexibility of the programs. Note that a group of the major players in the wireless device industry was formed in order to come up with a vendor-neutral standard based on XML for synchronizing user data (address books, appointments etc...) between different wireless devices\servers (<http://www.syncml.org>).

XML parsing creates extra CPU load and memory/storage overhead. Because the J2ME(CLDC\MIDP) devices are rather small in their memory and CPU power using XML should be done carefully (only when there is a good reason for doing so).

Parsing an XML document means retrieving the data that the document describes. The TAGS within the XML document define the structure and the boundaries of the embedded data elements. The following XML document describes a book. This XML document will be used along this chapter in all of its examples.

```
<?xml version="1.0"?>
```

```
<!DOCTYPE book
```

```
[
```

```
    <!ENTITY author "Haim Michael">
```

```
    <!ENTITY company "ZINDELL">
```

```
    <!ELEMENT author (#PCDATA)>
```

```
    <!ELEMENT title (#PCDATA)>
```

```
    <!ELEMENT pages (#PCDATA)>
```

```
    <!ELEMENT publishingDate (#PCDATA)>
```

```
    <!ELEMENT description (#PCDATA)>
```

```
<book>
```

```
    <author> &author; </author>
```

```
    <title> "The Israeli Guide to J2ME" </title>
```

```
    <pages> 540 </pages>
```

```
    <publishingDate> Jan 2002 </publishingDage>
```

```
    <description lang="en">
```

This book introduces, shortly & effectively, the Java 2 Micro Edition (J2ME).

The J2ME is the Java edition for small devices. The J2ME is divided into configurations and profiles. A configuration is designed for a specific kind of device based on memory constraints and processor power.

A profile is more specific than configuration. The profile is based on a configuration and adds more APIs to it. This book concentrates in the

CLDC

configuration (for very small devices such as cellular telephones and

paggers)

and the MIDP profile (designed for mobile information devices, such as

mobile

telephones).

```
    </description>
```

```
</book>
```

Generally speaking, the XML document has four main components:

elements

attributes

entities

DTDs

An element is usually two TAGS that comes together (starting tag & ending tag) to describe a piece of data. In the last XML document one of its elements is:

```
<pages> 540 </pages>
```

An attribute is usually placed within the starting TAG of a given element, and is used in providing additional information about the element. In the given XML document, the description starting TAG has the "lang" attribute with the value "en".

An entity is a virtual representation of a piece of data (either text data or binary data) that you can reference in the XML document. In the given example &author; is one of the entities that. When the XML document is parsed, each of its entities is replaced with its actual value.

A DTD (Document Type Definition) is an optional part of the XML document that defines the allowable structure for that document. #PCDATA within the DTD means "Parsed Character Data".

The parsers that enable parsing an XML document exist in two forms (types). One type of parsers is event-based interface and the other type is tree-based interface. Parsers that are event-based interface invoke on the object (the one that implements the needed interface) different methods according to the events that happen while parsing the document (element start, element end etc...). The SAX (Simple API for XML) is an industry standard event-based interface for XML parsing. A tree based XML parser reads the entire XML document and creates a graph of objects that represent the document data. Then it allows navigating between these objects and manipulate the data quickly and easily. The DOM (Document Object Model) is an industry standard tree-based interface for XML parsing.

The mobile devices usually have storage of just a few hundred kilobytes.

Therefore, lots of parsers that consume too much memory are not suitable for mobile devices. However, it is still possible finding small foot-print java based XML parsers that are good candidates for wireless device.

Among the small footprint java based XML parsers, the following parsers are the most popular ones:

TinyXML	http://www.gibardunn.srac.org/tiny/
NanoXML	http://nanoxml.sourceforge.net/
Aelfred	http://www.microstar.com/aelfred.html
KXML	http://kxml.enhydra.org/
MinML	http://www.wilson.co.uk/xml/minml.htm
Wbxml	http://www.trantor.de/wbxml/

Most of these parsers were developed using the J2SE. Therefore, before they can be used on mobile devices there are several changes that need to be done. This list of parsers include parsers that were originally written for the J2SE. The porting of most of these parsers to the J2ME(CLDC\MIDP) was already done by others.

TinyXML Parser for J2ME(CLDC\MIDP)

TinyXML is a non-validation you can download from

<http://gibardunn.srac.org/tiny/index.html>. A J2ME version of this parser can be

found at <http://www.kvmworld.com/Articles.TinyXML.shtml>. You should download the J2ME version of this parser.

In order to use this parser you first need to add the relevant class files to the **tmpclasses** directory in which the java byte code files of your source code is saved. Alternatively, you can simply add the jar\zip file with these classes to your class path. Then you should declare a class that extends the **HandlerBase** (a class that implements the DocumentHandler interface). This is the interface of all the event callback methods. The following example presents a simple use of the TinyXML parser. The following example includes two classes. The first class extends the **tinyxml.HandlerBase** so it is used for creating the listener object that listens to the XML events while the XML document is parsed. The second class is the midlet (extends the Midlet). Note that when you deploy the TinyXML on a real device you need to add the parser classes to the midlet suite. Moreover, if you don't know XML I suggest you read more material on XML before going on with the following example. A good book I can recommend is: "Professional Java XML" by wrox publishing house.

```
//fileelementName:XMLEventsParsing.java
```

```
//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.  
//All rights reserved. No part of the contents of this program may be  
//reproduced or transmitted in any firstForm or by any means without the  
//written permission of the publisher.
```

```
import javax.microedition.midlet.*;  
import javax.microedition.lcdui.*;  
import java.io.*;  
import javax.microedition.io.*;  
import java.util.*;  
import tinyxml.*;
```



```

public class XMLEventsParsing extends MIDlet implements CommandListener
{

    private Display display;
    private Form firstForm, secondForm;
    private Command exitCommand, backCommand, startCommand;

    public XMLEventsParsing()
    {
        //create the commands and the forms
        exitCommand = new Command("Exit", Command.EXIT, 1);
        backCommand = new Command("Back", Command.BACK, 1);
        startCommand = new Command("Start", Command.OK, 1);
        firstForm = new Form("XML Parsing");
        firstForm.addCommand(exitCommand);
        firstForm.addCommand(startCommand);
        secondForm = new Form("Results");
        secondForm.addCommand(backCommand);

        //setting the listeners
        firstForm.setCommandListener(this);
        secondForm.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
        display.setCurrent(firstForm);
    }

    public void commandAction(Command c, Displayable d)
    {
        //System.out.println("within commandAction");
        if(c==exitCommand)
        {
            //System.out.println("exitCommand was pressed");
            exit();
        }
        else
        {
            //2
            if(c==backCommand)
            {
                System.out.println("backCommand");
                display.setCurrent(firstForm);
            }
            else
            {
                //1
            }
        }
    }
}

```

```

        if(c==startCommand)
        {
            try
            {
                //creating a string buffer that will be used
                //for building the final message to the user
                StringBuffer xmlParsingResult = new
StringBuffer();

                //clearing secondForm
                System.out.println("startCommand");
                if(secondForm.size()==1)
                {
                    secondForm.delete(0);
                }

                //retrieving the data within J2MEbook.xml
                file
                Class theClass =
                Class.forName("XMLEventsParsing");
                InputStream is =
                theClass.getResourceAsStream("/J2MEbook.xml");
                StringBuffer sb = new StringBuffer();
                int temp = is.read();
                while(temp!=-1)
                {
                    sb.append((char)temp);
                    temp = is.read();
                }
                String str = sb.toString();
                System.out.println("The XML document to
be parsed is : ");

                System.out.println(str);

                // create a new XMLInputStream with the
                given test data
                XMLInputStream xmlStream = new
XMLInputStream(str);

                // create the xmlHandler
                EventsResponder xmlHandler = new
EventsResponder(xmlParsingResult);

                // create the parser
                XMLParser xmlParser = new XMLParser();

```

```

instantiated");

System.out.println("The XML parser was

// set your handler in order to receive events
from the parser

xmlParser.setDocumentHandler(xmlHandler);

// set the source to parse from
xmlParser.setInputStream(xmlStream);

System.out.println("parse() started");
xmlParser.parse();
System.out.println("parse() ended");

secondForm.append(new StringItem("xml
parsing result : ",

xmlParsingResult.toString()));

display.setCurrent(secondForm);
System.out.println("The result of the parsing
process is : ");

System.out.println(xmlParsingResult.toString());

    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
} //0
} //1
} //2
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void startApp()
{
    display.setCurrent(firstForm);
    //System.out.println("firstForm was set as the current screen");
}

```

```

    public void pauseApp()
    {
    }

    public void destroyApp(boolean cond)
    {
        display = null;
        firstForm = null;
        secondForm = null;
        exitCommand = null;
        backCommand = null;
        startCommand = null;
    }
}

```

```

class EventsResponder extends HandlerBase
{
    private StringBuffer sb;

    EventsResponder(StringBuffer sb)
    {
        this.sb = sb;
    }

    public void DocEnd()
    {
        System.out.println("==> DOCUMENT END");
        sb.append("END");
    }

    public void DocStart()
    {
        System.out.println("==> DOCUMENT START\n");
        sb.append("START");
    }

    public void ElementStart(String elementName, Hashtable attr) throws
ParseException
    {
        System.out.println("==> Element: " + elementName);
        sb.append("ELEMENT:"+elementName+"\n");

        if (attr != null)

```

```

        {
            Enumeration enumeration = attr.keys();
            while (enumeration.hasMoreElements())
            {
                Object obj = enumeration.nextElement();
                System.out.println(obj + " = " + attr.get(obj));
                sb.append(obj+"="+attr.get(obj));
            }
            System.out.println("");
        }
    }
}

```

When you declare the class that extends `HandlerBase` and override the inherited methods you should name them starting with a capital letter. For some reason, the `tinyxml.HandlerBase` was declared with method names that start with a capital letter. However, remember that the common design rule is declaring classes with names that start with capital letters.

The following are links for the source code of the NanoXML Parser classes, that the last example uses:

[DocumentHandler.java](#)

[HandlerBase.java](#)

[ParseException.java](#)

[XMLInputStream.java](#)

[XMLParser.java](#)

[XMLReader.java](#)

[CharacterUtility.java](#)

NanoXML Parser for J2ME(CLDC\MIDP)

NanoXML Parser was originally developed by Marc De Scheemaecker for using it on the J2SE. A J2ME version of this parser was developed by Eric Giguere. The original version of this parser (the JESE version) can be downloaded from <http://nanoxml.sourceforge.net/index.html>. More information about the J2ME version of this parser can be found at http://www.ericgiguere.com/microjava/cldc_xml.html. The NanoXML Parser is characterized in its very small size and simplicity. It is non-validating XML Parser, doesn't support DTD and entities and doesn't support mixed content. The NanoXML Parser consists of two sets of interfaces: one is an event-based interface and the other is a simple tree-based interface.

Parsing the document using the NanoXML Parser and its tree-based interface is rather simple. Object of the kXMLElement class is created and one of its methods is called in order to start the parsing process. After that, different methods can be invoked on the kXMLElement object (the one that has already been created) and return different information on the DOM hierarchy the XML document describes.

The following midlet makes a simple use of the tree-based interface the NanoXML provides.

```
//fileelementName:XMLNanoParsing.java
```

//Copyright (c) 2001 Haim Michael & Zindell Publishing House, Ltd.
//All rights reserved. No part of the contents of this program may be
//reproduced or transmitted in any form or by any means without the
//written permission of the publisher.

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.io.*;
import javax.microedition.io.*;
import java.util.*;
import nanoxml.*;
```

```
public class XMLNanoParsing extends MIDlet implements CommandListener
{
    private Display display;
    private Form firstForm, secondForm;
    private Command exitCommand, backCommand, startCommand;

    public XMLNanoParsing()
    {
        //create the commands and the forms
        exitCommand = new Command("Exit", Command.EXIT, 1);
        backCommand = new Command("Back", Command.BACK, 1);
        startCommand = new Command("Start", Command.OK, 1);
        firstForm = new Form("XML Parsing");
        firstForm.addCommand(exitCommand);
        firstForm.addCommand(startCommand);
        secondForm = new Form("Results");
        secondForm.addCommand(backCommand);

        //setting the listeners
        firstForm.setCommandListener(this);
        secondForm.setCommandListener(this);

        //getting the display object
        display = Display.getDisplay(this);
        display.setCurrent(firstForm);
    }

    public void commandAction(Command c, Displayable d)
    {
        //System.out.println("within commandAction");
    }
}
```

```

        if(c==exitCommand)
        {
            //System.out.println("exitCommand was pressed");
            exit();
        }
        else
        {
            if(c==backCommand)
            {
                System.out.println("backCommand");
                display.setCurrent(firstForm);
            }
            else
            {
                if(c==startCommand)
                {
                    InputStream is = null;
                    InputStreamReader isr = null;
                    try
                    {
                        //clearing secondForm
                        System.out.println("startCommand");
                        if(secondForm.size()==1)
                        {
                            secondForm.delete(0);
                        }

                        //retrieving an InputStreamReader to the
                        J2MEbook.xml file
                        Class theClass =
                        Class.forName("XMLNanoParsing");
                        is =
                        theClass.getResourceAsStream("/J2MEbook.xml");
                        isr = new InputStreamReader(is);

                        kXMLElement element = new
                        kXMLElement();
                        element.parseFromReader(isr);

                        secondForm.append(new
                        StringItem("children:",element.getChildren().toString()));
                        System.out.println(element.getChildren());
                        display.setCurrent(secondForm);
                    }
                    catch(Exception e)
                    {

```



```

        e.printStackTrace();
    }
    finally
    {
        if(is!=null)
        {
            try
            {
                is.close();
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }
    }
}

} //0
} //1
} //2
}

public void exit()
{
    destroyApp(true);
    notifyDestroyed();
}

public void startApp()
{
    display.setCurrent(firstForm);
    //System.out.println("firstForm was set as the current screen");
}

public void pauseApp()
{
}

public void destroyApp(boolean cond)
{
    display = null;
    firstForm = null;
    secondForm = null;
    exitCommand = null;
    backCommand = null;
    startCommand = null;
}

```

```
}
```

The following are links for the source code of the NanoXML Parser classes, that the last example uses:

[kXMLElement.java](#)

[XMLParseException.java](#)