F O R U M   N O K I A

Efficient MIDP Programming

Version 1.1; March 19, 2004

NOKIA

Contents

Change history

July 9, 2002	Version 1.0	Initial document release
March 19, 2004	Version 1.1	Document updated. Revision on January 16, 2007: Minor editorial changes including terminology update.

1 Introduction

This guide describes how to make your MIDlet more efficient. It covers the following aspects:

- Execution speed
- JAR file size (download costs, etc.)
- Use of resources (memory, networking)
- Perceived performance (user interface responsiveness)

This guide assumes that you are familiar with Java™ programming. It also assumes that you know the basics of MIDP programming; for instance, that you have read the Forum Nokia paper *MIDP 1.0: Introduction to MIDlet Programming* [MIDPPROG].

This guide focuses on MIDP performance issues, and covers only briefly the Java performance issues that are not specific to MIDP. For more information, see *Java Performance Tuning* [JAVAPERF] and *Java Platform Performance* [JAVAPP].

2 Execution speed

There's an old programmer's rule of thumb that programs spend 90% of their time in 10% of their code. Therefore, rather than trying to make all of your code efficient, you benefit far more by finding the "bottleneck" in your code and making it work more efficiently.

Another well-known principle is that careful design and algorithm choice yield greater benefits than line-by-line code optimizations. This is as true of MIDlets as it is of other programs, and it's easy to slow your MIDlet by making careless design choices.

Java programs usually spend a small part of their time executing your program code and most of their time executing the library code that you call. Therefore, get to know the performance of the libraries (especially graphics libraries), and choose carefully how you call them.

Remember that different phones' MIDP implementations vary significantly in their performance characteristics, so the best-performing approach on one phone may not be the best on another. For example, a particular library method may be implemented in Java on one phone, but wrap a faster native method on another.

Finally, note that performance measurements may vary not only between different phone models, but also between different versions of the same model. Manufacturers typically update software versions and sometimes even hardware components during a model's production lifetime.

2.1 Program execution speed

2.1.1 Measuring it

The usual tool for finding performance bottlenecks in a program is a *profiler*. Typically, however, you won't be able to run a profiler on a MIDlet running in a phone, and profiling a MIDlet running in an emulator may not tell you much, as emulators can have very different bottlenecks from actual phones.

The usual approach in MIDP is to take measurements yourself by adding a few extra lines to your program. To find out how long a call takes:

```
long startTime = System.currentTimeMillis();
doSomething();     // the thing you want to time
long timeTaken = System.currentTimeMillis() - startTime;
```

This will give you the time in milliseconds. To avoid varying results due to garbage collection during your test, you may want to call `System.gc()` before starting the test. To display the test measurements, use a special MIDlet screen or overwrite the results on the normal screen display.

Make sure to check the resolution of the phone's system clock. Its returned values may not have millisecond precision, so note if the returned value is, for example, always a multiple of ten, and make sure your test takes long enough that this isn't a problem.

2.1.2 Graphics operations

Typically, the execution speed of graphics operations is not a concern with MIDP's high-level screen classes like `Form` and `List`, but only with the low-level `Canvas` class, used for animations and games. The speed of `Canvas`'s graphics operations varies greatly between different phones, as it depends not only on the underlying hardware but also the efficiency of the phone's native graphics libraries. In Nokia's MIDP-enabled phones, you can draw thousands of short lines per second, or hundreds of rectangles or small images (like sprites).

If only a small part of the screen needs to change, you should request a repaint using the method:

```
Canvas.repaint(int x, int y, int width, int height)
```

This allows you to specify that only the changed area be repainted. Your `paint` method should then only repaint the area specified by its `Graphics` parameter's clip rectangle, possibly saving much calculation. However, be aware that if you issue repaint requests faster than the device can process them, it may merge several requests into one by calling the `paint` method with a clip rectangle covering all their rectangles; if the rectangles are widely spaced this will include much area that doesn't need repainting.

A different optimization that should work well on most phones is to use an off-screen image if your screen changes only slightly between repaints. Then you can draw the changes into the off-screen image and copy it to the screen in the `paint` method. Again, you only need to copy the area specified by the `Graphics` parameter's clip rectangle.

2.1.3 Garbage collection

Avoid creating unnecessary garbage objects on the heap. Often it is easy to reuse existing objects instead.

A particular problem is *immutable* objects — that is, objects whose state cannot be changed after creation. Immutable objects are widely used as they have great benefits for reliability of code and for thread safety.

However, immutable objects easily become garbage objects if their initial value is not needed, for instance if the value they represent changes. For each change, a new immutable object must be created with the new value, and the old one discarded to be garbage-collected. In MIDlets, often the advantages of immutable objects are not worth the cost, and it is better to make reusable objects that can be given new values.

The most familiar example is `java.lang.String`. Most programmers forget just how many normal uses of `String` involve creating garbage objects. The classic example is string concatenation; consider this function to reverse a string:

```
1:   static String reverse(String s)
2:   {
3:       String t = "";
4:       for (int i = s.length()-1; i >= 0; --i)
5:           t += s.charAt(i);
6:       return t;
7:   }
```

The assignment on line 5 does not modify string `t`, as `t` is immutable; instead, it creates a new string each time, copying the existing value and appending the new character. This method will unnecessarily create `s.length()` garbage objects. It is a classic example of the problem with immutable objects, but with strings there is a simple solution: the class `java.lang.StringBuffer` is a mutable partner to `String`, and the preceding example can be rewritten much more efficiently as:

```
1:   static String reverse(String s)
2:   {
3:       StringBuffer t = new StringBuffer(s.length());
4:       for (int i = s.length()-1; i >= 0; --i)
5:           t.append(s.charAt(i));
6:       return t.toString();
7:   }
```

Having said all this, often the advantages of immutability *are* worth the cost, and creating a few garbage objects is not a big deal if you're not creating thousands per second. Even a MIDP virtual machine can comfortably garbage-collect thousands of objects per second.

2.1.4 Multi-threading

Multi-threading can make your MIDlets perform much better, as one thread may be able to work while another is waiting on some condition (for example, waiting for an HTTP response, waiting for user input).

Remember that Java threading is not guaranteed to be pre-emptive, but may be cooperative. Therefore your code should not wait for a condition in a "tight loop," but should call `yield` or `wait` every time round the loop, for example:

```
try
{
    while (!stopped)
    {
        try
        {
            doSomething();
            synchronized(this)
            {
                wait(500);  //milliseconds, i.e. half a second
            }
        }
    }
}
catch (InterruptedException e)
{
}
```

Note that the `InterruptedException` is never in fact thrown in MIDP (since the method `Thread.interrupt` does not exist). However, for compatibility with other Java environments, the `wait` method still declares it as thrown, and hence an empty catch clause must be used.

2.2 Networking speed

2.2.1 Bandwidth and latency

Raw networking speed is usually measured in terms of *bandwidth* and *latency*, which are defined as:

- *Bandwidth*: the rate of data transfer in an open connection (usually measured in *bits per second*)
- *Latency*: the time taken for a single item of data to cross the network from the source to the destination

Both of these may have an *average* value and a *variation*; even if the average value is acceptable, if the variation is large, the user will frequently experience unacceptable values.

For large amounts of data, bandwidth usually has the most effect on networking speed. For small amounts of data, it is often latency that is more important.

Both measures depend on the network technology and on specific details of your network connection, so it is impossible to give precise numbers in general. However, to give you ballpark figures to help in making your designs, Table 1 presents some recent figures obtained from internal Nokia studies and informal tests.

	Bandwidth (kilobits per second)	**Typical first HTTP round trip latency**	**Typical following HTTP round trip latency**
GSM Circuit-Switched Data (CSD)	9.6 kbps	5 – 10 seconds	2 – 3 seconds
GSM High-Speed Circuit-Switched Data (HSCSD)	up to 43 kbps	5 – 10 seconds	2 – 3 seconds
GPRS	5 – 50 kbps	5 - 8 seconds (when network is congested, can be more)	2 – 4 seconds (when network is congested, can be more)

Table 1: Bandwidth and latency example figures

As you can see, these figures are somewhat slower than those you're used to from broadband Internet connections. In particular, the long latency rules out the possibility of highly interactive real-time multi-player arcade games, as you can't see and respond to other players' actions in real time.

One way to work around network latencies is to use asynchronous network operations (in a background networking thread). For instance, if you're sending a new high score to your high-score server, there's no need for the rest of the MIDlet to hang around waiting for that operation to complete; instead, you can leave the background networking thread to process it while you start playing the next game.

2.2.2 Minimizing HTTP round trips

Since HTTP latency is higher in wireless networks, try to minimize the number of HTTP round trips your MIDlet uses. Whereas an Internet Web browser will make many HTTP round trips to fetch different frames and images of a Web page, a MIDlet should aim to fetch everything it needs in one go.

If the MIDlet should gather data from several sources, one option is to use a proxy servlet to do the gathering, so that the MIDlet still only needs to make one request. For more about proxy servlets, see the following section.

2.2.3 Avoiding inefficient protocols

Complex XML-based protocols like SOAP can be very inefficient, with much overhead in terms of data size, parsing time, and parser code size. If you are designing an XML-based protocol for use with a MIDP client, try to keep it as simple as possible.

If you don't need the advanced capabilities of these protocols, you may be able to send the same information far more efficiently using a simple custom protocol (for instance `username=fred&password=secret`). Such simple custom protocols are described in the Forum Nokia documents MIDP 1.0: Introduction to Networked MIDlets [MIDPNET] and *MIDP 1.0: Fruit Machine Example* [FRUITMAC]. For debugging purposes it is a great help if the protocol is human-readable (this is why so many Internet protocols like HTTP are human-readable).

A case where you can usually use a simple custom protocol is when you are writing both the MIDlet client and the (for example, servlet) server. Beware of versioning — there may be a problem if you update the client and server to use a new protocol, but deployed MIDlets still exist that use the old protocol. It is a good idea to include a protocol version number in the opening communications between the client and the server.

If your MIDlet is talking to a server that you didn't write or don't control, or a server that must talk to clients other than your MIDlet, your MIDlet may have to talk XML or SOAP to it. Even in this case, you may be able to avoid this situation by using a proxy servlet:

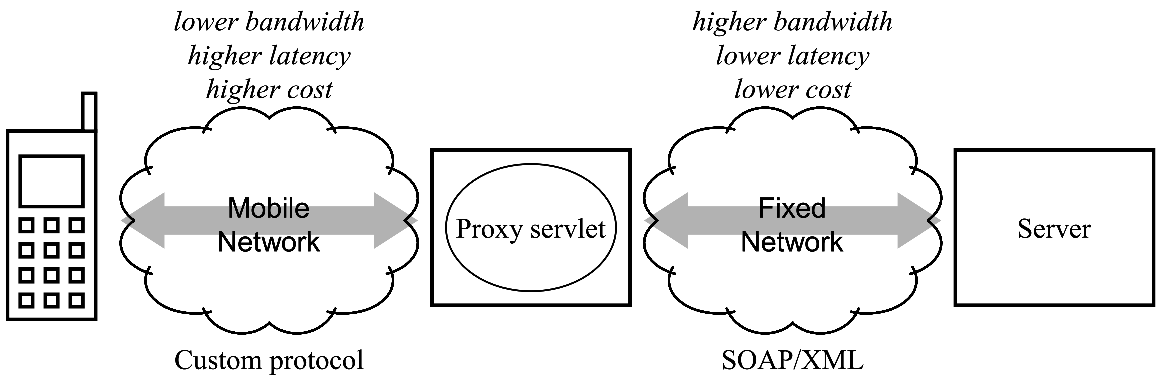

Figure 1: Proxy servlet

The proxy servlet converts between the SOAP or XML protocol and the MIDlet's custom protocol. It may make several SOAP or XML requests per MIDlet request — this is more efficient, since the connection between the proxy servlet and the server will be over a fast fixed-network connection.

2.2.4 Socket connections

MIDP 2.0 provides a number of interfaces intended to simplify UDP, TCP, and TLS based communication. A UDPDatagramConnection is used to send or receive UDP datagram messages. A ServerSocketConnection is used to accept inbound TCP connection requests. A SocketConnection is created for each accepted inbound TCP socket connection. A SocketConnection is also used for outbound TCP connections. Instead of making connection over HTTP, some helpful ideas are presented to designers to produce code with better performance and data throughput. These are based on real-life observations on the behavior of some existing MIDlets.

Sending and receiving data in very small pieces (one byte at a time or similar) is quite inefficient from the point of view of the underlying platform, the TCP protocol, and the bearer. It causes an unnecessary extra load to the platform and does not utilize the full capacity of TCP and the cellular network. Whenever possible, MIDlets should send and receive data in larger units.

Related to data receiving via TCP sockets, MIDlets should not use the input stream interface's `available()` method in a polling fashion, that is, query in a (busy) loop how much data is ready to be read. This is very inefficient and may cause a significant performance penalty. To be more efficient, the MIDlet should rather make more intelligent use of the stream reading methods that provide the possibility of specifying the amount of data to be read. The exact interface to use depends of course on the needs of the specific MIDlet.

3 JAR file size

There are several reasons to keep the JAR file size as small as possible:

- The MIDlet suite downloads faster, for example, GSM circuit-switched data takes about 1 second per kilobyte of JAR file
- Many WAP gateways are typically configured for small WAP pages and may not support large files (in particular, JAR files over 30 kB may hit this problem)
- Mass-market MIDP phones may have a very limited amount of storage space for MIDlet JAR files (this varies widely between phone models)
- Users will prefer small MIDlets because they can have more of them in their phone at once

3.1 Designing for small size

There are two sizes that matter: the size of the MIDlet suite's JAR file and the size occupied by the MIDlet suite when installed in the phone (if the JAR file is not just stored "as is"). The second size depends on the phone implementation, but the JAR size is a rather good predictor of this installed size.

Since the JAR archive format has individual headers for each class file, in general you're better off using as few classes as possible. Because of this, MIDlets will often be much less "object oriented" than normal Java programs. In particular:

- *Have one class per "thing"* – for example, don't separate something into Model, View, and Controller classes if one class can reasonably do it all
- *Limit use of interfaces* – an interface is an extra class that by definition provides no functionality; use interfaces only when needed to handle multiple implementations in the same delivered MIDlet
- *Use the "unnamed package"* – placing your MIDlet classes all in the same named package only increases the size of your MIDlet's JAR file; reserve the `package` statement for libraries
- *Consider using a source-code preprocessor instead of "static final" constants* – each such constant takes up space in your JAR file
- *Limit use of static initializers* – the Java class file format doesn't support these directly; instead, they are done using assignments at run time, for example, statically initializing an array of bytes ('`static byte[] data = {(byte)0x02, (byte)0x4A, …};`') costs 4 bytes per value, not 1 byte, in the resulting class file

3.2 Using an obfuscator

An obfuscator is a program that modifies your compiled Java program to remove all unnecessary information (like long method and variable names), making it hard to understand the result of "decompiling" it. As a protection method it is of less value for MIDlets, since they're so small that with some work even a decompiled obfuscated MIDlet can be figured out.

However, removing all of that unnecessary information makes your JAR file smaller, which is very helpful. The size reduction varies from obfuscator to obfuscator and from MIDlet to MIDlet, but tests on a few of our own MIDlets showed that a 10% reduction is typical. This is less than usually claimed for obfuscators, perhaps because MIDlets are small and there are some fixed overheads, and also because resources like PNG bit map files make up a larger proportion of a MIDlet's JAR file.

3.3 Libraries

In normal software development it is wise to develop and use libraries of frequently needed functionality. However, if the entire library is included in your MIDlet suite, you're likely to be paying for a lot of functionality you don't actually need.

You may do better to fall back on "cut and paste" reuse. It is a less efficient use of a developer's time, but a more efficient use of the JAR file space.

If you do use libraries, consider whether you need all the classes in the library. Perhaps you can remove several of the class files. You may even want to recompile some classes with unused methods removed.

Bearing this in mind, if you're writing a MIDP library you should aim to reduce dependencies between classes, so that unneeded classes can indeed be safely removed. Unfortunately, one typical approach to this is to use Java *interfaces*, which again expands your library size (see Section 3.1).

3.4 Keeping resources small

MIDlet suites often contain associated PNG bit maps, etc. Keep these as small and as few as possible.

There are significant differences in the size of a PNG bit map when saved with different bit map editing tools — not all optimize for size. Try a few of these tools and save with whichever gives the smallest result (even if you prefer to edit with another).

3.5 Combining image files

Just as it helps to minimize the per-file JAR overhead by having as few classes as possible, it also helps to have as few image files as possible. One often-used trick is to combine many images into one file:

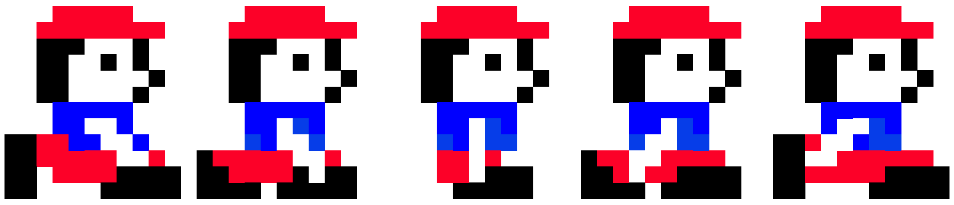

Figure 2: Combined image file

Once that large image has been loaded from the file, individual frames can be drawn as follows:

```
g.setClip(x, y, FRAME_WIDTH, FRAME_HEIGHT);
g.drawImage(fiveMenImage, x - FRAME_WIDTH * frameNumber, y,
            Graphics.TOP | Graphics.LEFT);
```

Here `frameNumber` is from 0 to 4; by cycling it in the sequence {0, 1, 2, 3, 4, 3, 2, 1} you will produce an animation of a walking man. If you do any more drawing in your `paint` method after the above lines, remember to change the clip window again as needed.

4 Use of resources

4.1 Heap memory

Typically, mobile phones do not have large amounts of heap memory. For instance, some of Nokia's first MIDP phones provide about 150 kB of MIDP heap memory. Some later models, such as the Nokia 6600 mobile phone, have as much as 9 megabytes of heap memory after rebooting the device and without other additional application running at the same time. In general, the situation with S60 devices is more complicated than with Series 40 devices, because it is possible to run several applications simultaneously. Methods like `Runtime.getRuntime().freeMemory()` and `Runtime.getRuntime().totalMemory()` don't give the correct values in S60 devices, because the amount of heap memory changes dynamically according to the current needs. Thus, it is not possible to give any guaranteed value for the heap memory amount.

Make sure that screens no longer needed (for example, splash screens) are released for garbage collection. Consider delaying creation of rarely used screens (for example, Help, Options) until they are used, and letting them be garbage-collected afterwards, each time. This will trade some loss of execution speed for extra heap memory.

Be particularly careful of memory leaks. These occur when an object is no longer needed, but there is still a reference to it somewhere (preventing it from being garbage-collected). If a reference to an unneeded object does not quickly go out of scope, remember to set it to `null`.

It is useful to understand how your MIDlet application uses heap memory. If you need to optimize your MIDlet's use of heap memory on some MIDP device, this understanding can help you find those places in the MIDlet code where reasonable changes could give the most significant heap memory savings. For example, the following approaches may help in some cases: use of more compact data representations for large data objects or collections, looking for large but sparsely populated arrays that can be more efficiently represented in other ways, or using more compactly encoded images.

4.2 Networking

MIDP devices are becoming mainstream around the same time as widespread use of packet data (for example, GPRS), so this is probably the network technology that your HTTP connections will run over. Therefore, your user will probably be paying per-packet or per-megabyte for the networking that your MIDlet does, and you should be conservative in spending your user's money.

Minimize the number and size of packets you send. For instance, try to only send packets when something changes, rather than sending a stream of "nothing happened" messages. If possible, avoid designs where your MIDlet has to send a constant stream of messages to a server, for instance by making MIDlet and server predict how the state changes, and only sending a message when the state differs from this prediction.

5 Perceived performance

The user actually cares less about how fast your MIDlet is, than about how fast your MIDlet *feels*. There are many well-known tricks to make the MIDlet feel faster, even though the tricks may actually slow it down a little.

5.1 Indicating liveness

Users notice very quickly when the MIDlet's display stops changing, and may suspect that the MIDlet or phone has crashed. If you do something that takes a long time, such as a big calculation or an HTTP network access, show a visible and animated indicator, for instance a `Gauge` as a progress indicator. The most popular Web browsers use good examples of progress indicators while they fetch a Web page.

5.2 Responsiveness

A MIDlet will feel unresponsive if it doesn't react quickly to users' key presses. Unfortunately, the performance textbooks give widely differing numbers for how fast counts as "quickly." For MIDlets, the normal speed of response will be that which the user experiences with the phone's native applications; this differs between phone models, but certainly anything longer than a second will feel unusually long.

To achieve quick responsiveness, make sure that your event call-backs (for example, `Canvas.keyPressed` or `CommandListener.commandAction`) return quickly, as they may be called from the same thread that redraws the screen. If they need to do some long action, start that in a separate thread, and consider having a way for the user to abort it.

Don't forget that it's not enough just to do something quickly; the user must *see* that something happened. Make sure that there is a visible (or audible) reaction to each key press.

5.3 Hiding delays

A favorite way to hide delays is to have a "splash screen," which is shown immediately while the rest of the MIDlet initializes itself. MIDP `Alerts` are good for splash screens, or you may use a `Form` if you want to implement a convenient call-back when the splash screen times out or is dismissed. The Forum Nokia Fruit Machine example [FRUITMAC] has such a splash screen.

The Fruit Machine example also features a useful trick for hiding its networking delay: it starts the animation of the spinning wheels before making its HTTP request to the server to find out what the result will be. The wheels continue to spin quickly until the HTTP response is received; then they gradually slow down and stop at the server's specified result. In this way the MIDlet seems to be very active, even when it is really just waiting for the server to respond.

6 Usability considerations

Usability is an important issue when designing applications, and it plays an even more important role in mobile devices with a small display and limited input mechanism. When thinking about usability, it must be taken into account that many situations can happen with the mobile device while the MIDlet is running. Those situations can be, for example, accidental power off, lost of network coverage, and incoming voice call.

6.1 Terminating the MIDlets

A MIDlet can be terminated due to many different reasons. Whether the reason occurs by purpose or by accident, the behavior of the MIDlet needs to be convenient for the user. Data needs to be stored to enable the MIDlet to start next time from the same situation. Table 2 presents events that can cause the MIDlet to be terminated. It also shows the behavior of the MIDlet and the exit point.

In case of a graceful exit in Nokia devices, the Java Application Manager (JAM) calls the MIDlet's `destroyApp()` method. This provides a perfect place to implement an auto-save mechanism for saving application data to RMS. User can then resume the same state next time. If the application shutdown is not completed in five seconds, the JAM kills the KVM immediately. This causes some limitations for storing data to the server or fetching something from the server.

Event	Notes	Show wait note	Type of exit	Exit point
Power off		No	Non-graceful	Power Off
Irresolvable error	For running out of memory, see next	Yes	Graceful	Application List
Java run-time memory exceeded	For other errors, see above	Error Note	Non-graceful	Application List
Natural exit	MIDlet's own exiting methods	Yes	Graceful	Application List
Back command	When Back is used for exiting	Yes	Graceful	Application List
Short press of END key		Yes	Graceful	Application List
Long press of END key		Yes	Graceful	Idle State

Table 2: Events terminating MIDlet

6.2 Application states

Many events can occur while the MIDlet is running, and it has to be remembered that the platform for which applications are being developed is a mobile phone. Table 2 presents the most common events and behavior that happens while the MIDlet is running.

Event	Notes	Behavior
Incoming SMS	Alert tone and vibration for incoming SMS. No other intervening effects	None
Incoming call	Note for incoming call is shown	MIDlet is running in the background while a note is displayed. After rejecting or accepting, the phone returns to the MIDlet display and execution continues instantly.
Notification for low battery	Note with alert tone	MIDlet is running in the background while a note is displayed. After a timeout, the phone returns to the MIDlet display and execution continues instantly.

Table 2: Events interrupting MIDlet

It has to be remembered that after the timeout of the notification or if the incoming call is accepted, the phone returns to the MIDlet display and execution continues instantly, and therefore an auto-pause implementation is required. By pausing the MIDlet, users can then resume back to the MIDlet whenever they are ready to do that. Because Nokia devices have not implemented the `PauseApp()` method in the MIDlet class, another approach has to be used.

Normally when some graphics is shown in a screen, it is created by using the `Canvas` class, which contains the `hideNotify()` method. This method is called shortly after the canvas disappears from the display, for example, when some note or an incoming call message hides the display. Using this `hideNotify()` method, a mechanism can be implemented, for example to stop running threads. The pause mode should be indicated to the user, for example, by painting a pause message or transition to the menu with Continue item.

The Canvas' `showNotify()` method is called prior to bringing the canvas back to display. Using this method, methods can be implemented for restarting threads or restoring the saved state, but it is important that the users have the possibility to make their own choice. Resuming should, therefore, happen after user action and not automatically.

A Displayable can be asked also if it is actually visible or not by calling its `isShown()` method. This is not an event driven action, so it requires monitoring. The example below shows one simple way to implement an auto-pause method().

`Canvas` class:

```
public void hideNotify() {
  pauseOnCanvas(); }

public void showNotify() {}

public void pauseOnForm() {
  canvasPaused = false;
  stop();
  myForm.pauseForm();
  parent.setDisplayable(myForm); }

public void pauseOnCanvas() {
  if (myForm.pauseOnForm() == true)
    pauseOnForm();
  else {
    paused = true;
    stop();
    repaint(); }
}

public void resumeCanvas() {
  canvasPaused = false;
  start();
}
```

`Form` class:

```
public void pauseForm() {
  addCommand(cmResume);
  formPaused = true;
}

public void resumeForm() {
  removeCommand(cmResume);
  formPaused = false;
  parent.setDisplayable(myCanvas);
  myCanvas.resumeCanvas();
}
```

7 References

[FRUITMAC] *MIDP 1.0: Fruit Machine Example*, Forum Nokia, 2004, http://www.forum.nokia.com

[JAVAPERF] *Java Performance Tuning*, Jack Shirazi, O'Reilly, 2000, ISBN: 0-596-00015-4

[JAVAPP] *Java Platform Performance: Strategies and Tactics*, Steve Wilson and Jeff Kesselman, Addison-Wesley, 2000, ISBN: 0-201-70969-4

[MIDPNET] *MIDP 1.0: Introduction to Networked MIDlets*, Forum Nokia, 2004, http://www.forum.nokia.com

[MIDPPROG] *MIDP 1.0: Introduction to MIDlet Programming*, Forum Nokia, 2004, http://www.forum.nokia.com

8 Evaluate this resource

Please spare a moment to help us improve documentation quality and recognize the resources you find most valuable, by rating this resource.