

Programming the MIDP Lifecycle on Symbian OS

Author: Martin de Jode

Version: 1.00

Date: November 2004

1. Introduction

The Mobile Information Device Profile (MIDP) has become an increasingly popular programming environment on mobile phones. Every MIDP application (MIDlet) should correctly handle the MIDlet lifecycle state changes. However, the [MIDP specification](#) and javadoc is perhaps not as precise as it could be in explaining how application developers should achieve this. Symbian has always provided its own implementation of MIDP and in this paper we give Symbian's interpretation of the MIDP lifecycle and show how developers should correctly manage lifecycle changes when programming MIDP on Symbian OS.

2. MIDP Lifecycle States

The Mobile Information Device Profile (MIDP) specifies a lifecycle model for MIDlets. A MIDlet can be in any one of three states: ACTIVE, PAUSED or DESTROYED.

ACTIVE

The MIDlet is executing normally

PAUSED

A quiescent state - memory and CPU consumption should be minimised. The MIDlet should not be holding any shared resource, and no non-essential threads running.

DESTROYED

The MIDlet has terminated and should be eligible for garbage collection.

3. Moving the MIDlet through states

The abstract MIDlet class (which all MIDP applications must extend) defines methods allowing the Application Management System (AMS) to control the creation, starting, pausing and destruction of MIDlets. The AMS is a state machine responsible for (among other things) moving the MIDlet between states by calling one of the following methods as appropriate.

```
startApp  
pauseApp  
destroyApp
```

It is the responsibility of the AMS state machine to maintain the MIDlet state and ensure the lifecycle methods are called appropriately.

As far as the AMS is concerned the MIDlet moves into the ACTIVE state when `startApp` is invoked. The AMS moves the MIDlet to the PAUSED state when the `pauseApp` method returns and similarly when `destroyApp` successfully returns the MIDlet is in the DESTROYED state.

It is the responsibility of the application developer to implement the above methods appropriately in the concrete MIDlet class, so that the application takes the appropriate action in response to a state change. How the developer implements `startApp`, `pauseApp` and `destroyApp` is obviously application specific, but a few guidelines are worth starting.

- well-behaved applications should always provide non-trivial implementations for `startApp`, `pauseApp` and `destroyApp`.
- `startApp` may be called more than once, so it is not the place for any “one off” initialisation.
- `pauseApp` should minimise resource consumption, release any shared resources and stop any non-essential threads.
- `destroyApp` should facilitate graceful termination of the application, saving persistent data to storage, releasing all resources and terminating all running threads

It is also helpful to understand the circumstances in which the AMS calls these methods. On Symbian OS the behaviour of the AMS is customisable by the handset manufacturer (licensee) so the circumstances under which the notification methods are called (particularly the `pauseApp` method) may (and does) vary from device to device. A few guidelines however are given below.

- `startApp` is called by the AMS after initial creation of the MIDlet, and may be called after a previously active MIDlet has been paused. For instance on Symbian OS it will be called after a previously backgrounded MIDlet that is in the PAUSED state is moved into the foreground.
- Do not assume `pauseApp` is automatically called when a MIDlet is backgrounded (loses focus). On Series 60 `pauseApp` is not automatically called when a MIDlet is backgrounded, whereas on UIQ devices the AMS should call `pauseApp` when the MIDlet moves to the background
- `pauseApp` will typically be called by the AMS on a backgrounded MIDlet that is not in the PAUSED state, when the AMS is notified by the operating system of low battery conditions.

- `destroyApp` will typically be called by the AMS when the operating system notifies the AMS of low memory conditions or power down (switch off). `destroyApp` may also be called by the AMS in response to user action to kill a running application (for instance on Series 60 using the “Menu” key to bring up running applications, then the “Clear” key to terminate the selected application).

4. Requesting a state change

A MIDlet can initiate state changes itself. When a MIDlet wants to request a state change it can do so itself by invoking one of the following methods:

```

notifyPaused
notifyDestroyed
resumeRequest

```

The first two methods notify the AMS that the MIDlet is voluntarily entering the PAUSED or DESTROYED state respectively and tells the AMS to update the MIDlet’s state. The final method indicates to the AMS that a MIDlet in the PAUSED state is interested in entering the ACTIVE state.

Now let us look in more detail at how these methods are employed. If the MIDlet wants to terminate itself (as opposed to being moved to the DESTROYED state by the AMS via a call to `destroyApp`) it should perform all necessary clean-up and then call `notifyDestroyed`. On receiving a `notifyDestroyed` notification the AMS state machine will assume the MIDlet is ready to be terminated and change the state of the MIDlet to DESTROYED. The AMS will not call `destroyApp` after a `notifyDestroyed` notification.

Many standard texts (and indeed IDE templates) incorrectly give example code such as

```

public void commandAction(Command c, Displayable d) {
    ...
    if (c == exitCommand) {
        destroyApp();
        notifyDestroyed();
    }
    ...
}

```

The rationale for the above is that typically the clean-up that needs to be performed prior to calling `notifyDestroyed` is identical to the clean-up performed in the implementation of the `destroyApp` method. However, a strict interpretation of the MIDP specification would predicate against this style. `destroyApp` is a call-back method that is invoked by the AMS to indicate to the MIDlet that it is going to be destroyed by the AMS. Just

as one would not call `actionPerformed` in AWT directly, or `commandAction` in MIDP directly, the MIDlet should not directly call `destroyApp`. A more correct and robust style is shown below.

```
public class MyMIDlet extends MIDlet... {
    private final static int PAUSED = 0;
    private final static int ACTIVE = 1;
    private final static int DESTROYED = 2;

    private int stateTracker = PAUSED; //MIDlets initialized into the PAUSED state

    ...

    public void commandAction(Command c, Displayable d) {
        //commit suicide
        if (c == exitCommand) {
            performCleanup();
            notifyDestroyed();
        }
        ...
    }

    private synchronized void performCleanup( ) {
        if ( stateTracker != DESTROYED ) {
            //do what's necessary
            ...
            stateTracker = DESTROYED;
        }
    }

    public void destroyApp(boolean unconditional){
        performCleanup();
    }
}
```

When the MIDlet wishes to terminate, in this example as a result of the user selecting the exit command, first the `performCleanup` method is called. This method is synchronized because it can also be called on a different thread by the AMS via the `destroyApp` method. The `performCleanup` method makes use of a “housekeeping” variable named `stateTracker`, that tracks the current state of the MIDlet. We do this because the actual state of the MIDlet is maintained by the AMS, and is not directly accessible to application code. The `performCleanup` method first checks the `stateTracker` variable and provided that the MIDlet has not already been moved to the DESTROYED state (by the AMS via `destroyApp`) any necessary clean-up is then performed. Finally the `stateTracker` variable is set to DESTROYED to prevent the clean up code being run again. Upon the return from

the `performCleanup` method the MIDlet calls `notifyDestroyed`, indicating to the AMS that the MIDlet is ready to enter the DESTROYED state. Note that the implementation of the AMS is such that if the MIDlet is already in the DESTROYED state, calling `notifyDestroyed` has no effect.

You may well ask what the difference is between the two approaches. The answer is that it is ultimately the responsibility of the AMS to maintain the state of the MIDlet and therefore issue calls to `startApp`, `destroyApp` and `pauseApp` as appropriate. If the MIDlet starts calling these methods directly then potentially the MIDlet may end up in an inconsistent state. For instance what happens if the AMS has decided to move the MIDlet to the PAUSED state via a call to `pauseApp`, whilst the MIDlet is calling `destroyApp` itself? Or indeed if the AMS calls `destroyApp` while the MIDlet is also calling `destroyApp`. In the latter case at the very least `destroyApp` will get called twice. So not only is the first example bad style, it could also have unpredictable consequences depending on the implementation of `destroyApp` and/or `pauseApp`.

Now we'll look at how the MIDlet can initiate a transition to the PAUSED state. If the MIDlet itself wants to enter the PAUSED state voluntarily (as opposed to being moved to the PAUSED state by the AMS via a call to `pauseApp`), it should release any shared resources, move into a quiescent state and then call `notifyPaused`. Again note on receiving a `notifyPaused` notification the AMS will assume the MIDlet is ready to enter the PAUSED state and it will not invoke `pauseApp` itself.

A common scenario in which a MIDlet might want to move to the PAUSED state is when the MIDlet is backgrounded. For games normally there is not really much point in the application continuing to run in the background, draining the battery, whilst unseen the Aliens continue to zap the Space Ships. Very often the right thing to do will be to go into the PAUSED state on being backgrounded. As we saw earlier the MIDP specification does not define how the AMS should behave when moved into the background. The behaviour is implementation dependent and UIQ and Series 60 take different approaches. So how does the developer ensure consistent behaviour across platforms?

If the developer wishes to ensure that the MIDlet moves into the PAUSED state in response to being backgrounded then the developer should use the `Displayable.isShown` method or the `Canvas.showNotify` and `hideNotify` methods to monitor the state of the display, and take appropriate action when the MIDlet is backgrounded. For example some skeleton code for handling a MIDlet Canvas being backgrounded is given below.

```
public class MyMIDlet extends MIDlet... {
    ...
    private int stateTracker = PAUSED; //MIDlets initialized into the PAUSED state
    ...
}
```

```

public startApp() {
    stateTracker = ACTIVE;
    //do whatever is necessary to activate
    ...
}

public synchronized void prepareToPause( ) {
    if ( stateTracker == ACTIVE ) {
        //do what's necessary to passivate
        ...
        stateTracker = PAUSED;
    }
}

public void pauseApp(){
    prepareToPause();
}
}

public class MyCanvas extends Canvas {
    private MyMIDlet midlet;

    public void showNotify( ) {
        midlet.resumeRequest( );
    }

    public void hideNotify( ) {
        midlet.prepareToPause( );
        midlet.notifyPaused();
    }
}

```

Let us look at this code in more detail. The first thing the `startApp` method does is set the `stateTracker` variable to `ACTIVE`, since the `MIDlet` becomes `ACTIVE` as soon as the `startApp` method is entered (as the `MIDlet` may start acquiring shared resources and launching threads). Access to the `stateTracker` variable isn't synchronised here since `startApp` should only ever be called by the AMS.

When the `MIDlet Canvas` is backgrounded the system invokes the `hideNotify` method. The implementation of `hideNotify` for `MyCanvas` invokes the `prepareToPause` method. This method is synchronized since it may also be called (from a different thread) by the AMS via the `pauseApp` method. The `prepareToPause` method checks the `stateTracker` variable and provided the `MIDlet` is `ACTIVE` (i.e. not already paused or destroyed) it performs the necessary instructions to move the `MIDlet` to a quiescent state and finally sets the `stateTracker` variable to `PAUSED`. Once the `prepareToPause` method returns the `notifyPaused` method is then invoked to

notify the AMS that the MIDlet wishes to be moved to the PAUSED state. Note that if the MIDlet is not in the ACTIVE state then `notifyPaused` has no effect on the AMS state machine.

When a previously backgrounded MIDlet Canvas is moved to the foreground, the system invokes the `showNotify` method. The implementation of the `showNotify` method in this example simply invokes the `resumeRequest` method, indicating to the AMS that the MIDlet is interested in entering the ACTIVE state. If the AMS can fulfil this request it will invoke the `startApp` method. Note that the implementation of the AMS is such that if the MIDlet is already in the ACTIVE state, `resumeRequest` has no effect.

The skeleton code above is provided as a trivial example of the correct way to manage state change requests – rather than template code. You, as the developer, will need to adapt your implementation to your application's specific requirements. The key point to get over is that your MIDlet code has no business calling `startApp`, `pauseApp` or `destroyApp`, even if it appears to be harmless to do so.

5. Discussion

One of the sources of ambiguity in the MIDP specification is to which entity maintains the MIDlet state. This arises because the AMS can move the MIDlet between states via `startApp`, `pauseApp` and `destroyApp`, whilst the MIDlet itself can also initiate a state change via `notifyPaused`, `notifyDestroyed` or `resumeRequest`.

For example the javadoc comments for the MIDlet class state the following:

The application management software maintains the state of the MIDlet and invokes methods on the MIDlet to notify the MIDlet of change states...

Whereas elsewhere the javadoc for the `notifyDestroyed` method has the following comments:

Used by a MIDlet to notify the application management software that it has entered into the Destroyed state...

This leads to some confusion as to which entity is responsible for maintaining the MIDlet state; the MIDlet or the AMS. The answer is that it is the AMS that calls the shots, in that it is the AMS that is responsible for moving the MIDlet between states and maintaining the record of the MIDlet state. For example as far as the AMS is concerned the MIDlet is in the DESTROYED state either after an invocation of `destroyApp` returns successfully or upon receipt of a `notifyDestroyed` notification, regardless of whether the application has actually released any resources or terminated its running threads. Once the AMS has registered the MIDlet as in the DESTROYED state, the MIDlet will be eligible for garbage collection and if no other MIDlets are running, the current instance of the VM will be terminated, irrespective of whether the MIDlet is ready for termination or not.

It is worth discussing what happens if the MIDlet does not manage the state changes correctly. If the MIDlet holds onto resources despite being moved to the DESTROYED state by the AMS, and there are other MIDlets running in the VM, then these resources will remain held until the VM finally terminates (when all other MIDlets have been moved to the DESTROYED state). If there are no other MIDlets running then termination of the VM will free up resources even if the MIDlet hadn't done so, but obviously any data that needed to be stored persistently will be lost. In the case of the MIDlet failing to manage the transition to the PAUSED state the consequences may be threefold: firstly the MIDlet may present a poor user experience, particularly in the case of interactive games that fail to pause when backgrounded; secondly if the MIDlet fails to move into a quiescent state and free up resources when requested to do so by the operating system via the AMS, the AMS may then be forced to take more drastic action and terminate the MIDlet completely; finally if, when moved to the PAUSED state, the MIDlet continues to hold onto shared resources, for example the camera, these resources will not be available to other applications whether native or Java.

To ensure the MIDlet behaves appropriately in response to a state transition it is obviously useful for the MIDlet to maintain an internal record of its state (as we did in the earlier skeleton code by way of the `stateTracker` variable) thus ensuring the MIDlet behaves in a manner consistent with the expectations of the AMS. For more information on how to do this rigorously see the White Paper [Managing the MIDlet Life-Cycle with a Finite State Machine](#), available from Sun's Developer Network website.

6. Summary

Implementing the MIDlet lifecycle model is the source of some confusion. In this paper we have presented Symbian's interpretation of the MIDP lifecycle and indicated how developers should correctly program the lifecycle methods on the Symbian OS Java Platform.

For more information on programming MIDP see the [resources](#) listed below.

7. Resources

[MIDP 2.0 Specification - JSR 118](#)

Java Community Process, Java Specification Request 118.

["Managing the MIDlet Life-Cycle with a Finite State Machine"](#)

White Paper, Sun Developer Network.

["Programming Wireless Devices with the Java 2 Platform, Micro Edition \(2nd Edition\)"](#)

Roger Riggs et al, Addison Wesley, 2003.

[“Programming Java 2 Micro Edition on Symbian OS: A developer’s guide to MIDP 2.0”](#)

Martin de Jode, Wiley, 2004

[Back to Developer Library](#)

Want to be kept informed of new articles being made available on the Symbian Developer Network?

[Subscribe to the Symbian Community Newsletter.](#)

The Symbian Community Newsletter brings you, every month, the latest news and resources for Symbian OS.