Writing Portable J2ME Applications

Author: Anurag Gupta
Mindfire Solutions, www.mindfiresolutions.com

Table of Contents

1. **Introduction…….. 3**

2. **Target audience……..3**

3. **J2ME Porting Guidelines .....……....3**

4. **JPP Introduction…….....……....7**

1. Introduction

It is hardly the case that an application built for one mobile device will not be ported to other handsets.

Handsets vary significantly in various parameters like:

- screen size (for ex SE-S700i - 240X320, Nokia 7250i - 128X128),
- heap size,
- maximum application size

Hence, there is a need to follow certain guidelines, which can help you in making your application (esp games) porting friendly.

In this article, we would be discussing such points and give an introduction to Java Pre Processor.

2. Target Audience

This document is intended for J2ME developers involved in application and game development and who are involved in handset porting for the same.

3. J2ME Porting Guidelines

We are putting forth some practical points, which if followed may help you in making your game porting friendly.

I. Using "Switch-Case"

```
Public static final int STATE_RUNNING = 1000;
Public static final int STATE_PAUSED = 2000;
Public static final int STATE_RESUME = 3000;

Switch(n)
{
    case STATE_RUNNING:
            dosomething2();
    break;

    case STATE_PAUSED:
            dosomething1();
    break;
}
```

There's nothing wrong in the above statement but it would be better if we use the smaller integer values like:

```
Public static final int STATE_RUNNING = 1;
Public static final int STATE_PAUSED = 2;
Public static final int STATE_RESUME = 3;
```

II. Methods

Following is the list of the restriction keywords applicable for methods starting from the slowest one in terms of performance:

- synchronized
- interface
- instance
- final
- static

So try to use the *final* and / or *static* methods wherever possible.

III. Arguments in the methods

Calling the methods and passing arguments in methods cause overheads, which finally affect the performance of your application. So try to minimize the number of methods in your application.

IV. Accessing the Arrays

Instead of accessing an array element a number of times, store that element in a local variable and use it.

V. Using the device specific API

Try to avoid device specific APIs as far as possible. But you do come across situations when it cannot be avoided; for example, to get the full screen display on a nokia phone you need to extend your canvas class from *FullScreen* class.

VI. Separate Game Logic from View

It's always advisable to write game logic in a separate class and views in separate classes. This way you would not need to change the game logic when the game is to be ported to other devices. In most cases, changing the graphics does the trick, which can easily be handled by view classes.

VII. Use of shift operators for multiplication and division

Using Bit shift operations is a faster way to do the multiplication (left shift) and division (right shift) by 2.

VIII. Garbage Collector

Avoid calling the garbage collector too often in your application as this result into serious performance issue. Garbage collection process takes a lot of time and resources and may cause your application to hang. Its better to leave garbage collection to KVM and to write your application so that you free your objects (set them to NULL) soon as their utility is over.

IX. Instance Variables

Generally it's a very good practice of keeping your variables private and writing the setters and getters for accessing them. But NOT in J2ME.
The methods generate byte code, which ultimately increases your application size. Apart from overhead of calling methods, many devices have very small application sizes, for ex in older Nokia Series 40 devices you have the application size limited to only 64 kb. Hence it's advisable to use public variables if you can afford to do that.

X. Number of Classes

Try to minimize the number of classes in your application. More the number of classes, more overhead on the KVM and also more is the application size.

XI. Using "setClip"

There might be few areas in your game's view that are static, that is they don't change with time or actions. So we should paint only those areas which need to be refreshed not the whole screen. You can easily do that by using the **setClip** method, which clips the area that you want to be refreshed. Painting is very time costly process and the more efficiently you can use it, the faster your game can be.

XII. Object Creation

Try to minimize the creation / deletion of objects in your application. Creating objects is a memory consuming process. Also continuous creation & destroying of objects causes memory fragmentation that ultimately leaks memory that cannot be reused unless KVM (and hence device) is restarted (known devices with problems: Nokia 3650, 3660).
So it's better to create the object pool and then use the objects from that pool.

XIII. Packages

Don't use the packages in your class structure and put all your classes in the default package. This would reduce the processing needs and may also keep the application size slightly on lower side.

XIV. Avoid deep hierarchies in your class structure

Since in java, the object binding is done at run time, which is a slow process, it's better not to use the deep hierarchies in your class structure.

XV. Obfuscation

Obfuscation is a technique that helps compress your application and make it secure against hacking at same time. It shortens your variables & method names, removes the repetitive code and compresses the graphics. Since the class files can be decompiled, decompiling obfuscated files is not of much help as the code has been jumbled up so much that it will take a lot of time to know what exactly is written in the file. Until now, obfuscation is a one-way process.

XVI. Using appropriate data types

Since in the mobile devices, heap is a major factor so you must declare your variables with appropriate data types. For example when you know that the variable's value will not exceed 128, define it as a byte rather than an int or short.

4. Introduction to JPP

JPP stands for Java Pre Processor. It is actually used to convert a ".jpp" file into a ".java" file. ".jpp" file contains java code with some additional predefined configuration tags used to segregate code for various devices. So when you process this ".jpp" file with a bat file having the particular configuration setup, it will spit out a proper java file without the tags customized for that particular configuration.

Why to Use JPP?
Since your application needs to be ported across various devices and each one has its own specification that may lead to having separate code base for each device. In such a situation if you need to change any feature in the application, you need to change it in all the codebases. That is where this JPP comes in handy. You have a single codebase (".jpp" files) with multiple configurations added to support the various devices. Now you can just change the feature in ".jpp" file, run the bat file and spit out the java files for various devices without additional coding.

Example:

GameUICanvas.jpp

```
#ifdef NOKIA3650
import com.nokia.mid.ui.FullCanvas;
#endif
import javax.microedition.lcdui.*;
#include "defines.jh"

final class GameUICanvas extends
#ifdef NOKIA3650
    FullCanvas
#endif
{
    /* the screen which is currently
     * being displayed*/
    UICanvas currUI;

    /**
     *Constructor
     */
    public GameUICanvas()
    {
            currUI = null;
    }

    public void hideNotify()
    {
            if(currUI != null)
            {
```

```
                currUI.hideNotify();
        }
}

public void showNotify()
{
        if(currUI != null)
        {
                currUI.showNotify();
        }

}

/**
 * <code>paint</code>
 * This is the overridden method of
 * the canvas class which is used for
 * painting on the screen.
 */
public void paint(Graphics g)
{
        if (currUI != null)
                currUI.paint(g);
}


/**
 * <code>keyPressed</code>
 * This is being called by the
 * canvas class when the user
 * presses any key.
 * @param keyCode. The key code of the
 * key pressed.
 */
protected void keyPressed(int keyCode)
{
        if (currUI != null)
                currUI.keyPressed(keyCode);
}

/**
 * <code>callPaint</code>
 * for calling the repainting
 * of the screen.
 */
protected void callPaint()
{
```

```
                repaint();
                serviceRepaints();
        }
    }
```

defines.jh

```
#ifdef NOKIA3650
#include "defines3650.jh"
#endif
```

defines3650.jh

```
#define ID_SPLASH_LOGO          1
#define ID_START_SCREEN         2
#define ID_MAIN_MENU        3
#define ID_OPTIONS_MENU         4
#define ID_VIEW_MENU        5
#define ID_SOUND_MENU       6
```

bat file

```
jpp GameUICanvas.jpp /nc /noremove /DNOKIA3650 /DRELEASE
/DSeries60Midp1
```

GameUICanvas.java

```
import com.nokia.mid.ui.FullCanvas;
import javax.microedition.lcdui.*;
final class GameUICanvas extends FullCanvas {

    UICanvas currUI;

    public GameUICanvas()
    {
            currUI = null;
    }

    public void hideNotify()
    {
            if(currUI != null)
            {
                    currUI.hideNotify();
```

```
                }
        }
        public void showNotify()
        {
                if(currUI != null)
                {
                        currUI.showNotify();
                }
        }
        public void paint(Graphics g)
        {
                if (currUI != null)
                        currUI.paint(g);
        }
        protected void keyPressed(int keyCode)
        {
                if (currUI != null)
                        currUI.keyPressed(keyCode);
        }
        protected void callPaint()
        {

                repaint();
                serviceRepaints();
        }
    }
```

Above example shows the various participants when you use the JPP.

- In the ".jpp" file surround the phone specific code inside the custom tags, which we use later.
- Create ".jh" (java header) files which have phone specific constants which are used in the ".jpp" file.
- Create phone specific bat files with command line parameters specifying which tags to use (same as used in JPP file).
- Run the bat file. This will give you as output the Java file which is specific for a phone.

There is no substitute for practical experience. In time you will discover your own secrets about the performance characteristics for a particular device you are working with. So take the "porting" demon head on. Hope points listed down in this document will help you. Incase you come across more characteristics which can be used by others, do share them with developers across the world.