



There are three key factors to considered when you want to optimize your Java ME applications

- Performance
- Size

You should delay optimization until the last minute, after you create the main features of our application, but you need to keep in mind all the key factors throughout the development cycle to avoid huge changes at the end.

Contents

- 1 Performance
 - ◆ 1.1 Threads
 - ◆ 1.2 System Callbacks
 - ◆ 1.3 Different Devices - Different Performances
 - ◆ 1.4 I/O
 - ◆ 1.5 General
- 2 Size
 - ◆ 2.1 Obfuscator
 - ◆ 2.2 Images

Performance

Rule number one for performance is "**keep it simple**", don't try to create over complex systems, for your mobile phone, remember people want fast and easy to use applications/games, to use on the move. With this mind take in account the following issues:

Threads

- Use only one application thread, avoid multiple threads. Many devices cannot handle too many threads, they simple stop. Network thread is the only exception for creating a new thread
- Minimized usage of synchronized, is expensive on legacy devices and is very common to create poorly synchronized code. Synchronizing run with paint() and keyPressed() is ok.
- Avoid using Timer class, an extra thread is created for each one.
- Create a background thread in startApp() and reuse it.
- Don?t use Display.callSerially(), is very slow and buggy on many devices. A new thread is created in most implementations
- Avoid serviceRepaints() on legacy devices when performance become an issue
- Might need to ensure thread safety (background thread and system thread) if serviceRepaints() is not used

An example for the base structure for you application:

```
public void startApp() {  
    animationThread = new Thread(this);  
}
```

```
public void run() {
    init();
    while(!exitApp) {
        updateModel(); // your app logic
        repaint();
        serviceRepaint();
        sleep(50); // may choke slow devices if too small
    }
}
```

System Callbacks

You must handle with care the system callbacks, they are called by the system thread. They must not block and should return as soon as possible to avoid slowing down the VM. A crash can happen if not returning quick enough. Here is a list with the more common system callbacks:

- * paint
- * keyPressed
- * startApp/pauseApp
- * hideNotify/showNotify
- * MIDlet constructor

Below you have a bad example of a system callback:

```
public void paint(Graphics g) {
    updateModel();
    drawBackground(g);
    drawForeground(g);
}

public void run() {
    while(!exitApp) {
        repaint();
        serviceRepaints();
        sleep(50);
    }
}
```

As you may have noticed we are updating our model, an expensive operation, during the system event paint. We should change our code to the following:

```
public void paint(Graphics g) {
    drawBackground(g);
    drawForeground(g);
}

public void run() {
    while(!exitApp) {
        updateModel(); // better do it here
        repaint();
        serviceRepaints();
        sleep(50);
    }
}
```

Different Devices - Different Performances

There are huge differences between high end and low end devices, devices can be much faster or slower than you expected, so you cannot assume performance.

If you have your game logic based on frame rate this may result in totally unplayable game. You should instead base your game on actual time instead, like the example below:

```
public void updateModel() {
    curTime = System.currentTimeMillis();
    elapsedTime = curTime - prevTime;
    // using elapsedTime for your app logic here
    prevTime = curTime;
}

public void run() {
    while(!exitApp) {
        updateModel();
        repaint();
        serviceRepaints();
        sleep(50);
    }
}
```

Another thing to take in account is the performance of some APIs, some calls are slower than others:

- drawString(), replace with graphics font
- drawArc(), Vector, Hashtable, replace with own implementation
- drawImage(), replace large images with a series of smaller images for some devices

Another thing to handle with care is collision detection, it must be calculated in between frames to avoid problems on low frame rate devices.

Disable certain features on slow device via JAD entry. E.g. if "tree.png" does not exist, don't draw the tree sprite

I/O

The use of RMS and getResourceAsStream calls is very slow. Take the following consideration when using them:

RMS

- Read entire record into a buffer
- Then parse the buffer
- Similarly, write to a single buffer, then write the buffer to a record

getResourceAsStream()

How_to_optimize_in_Java_ME

- Extremely slow on some devices (as slow as 20 bytes per second)
- Class loading is much faster on these devices
- Workaround?store data in separate class files instead of using resource. This may result in slightly longer application load time

General

Here some ?Little things? that make big differences:

- Avoid unnecessary object creation/memory allocation.
- Reduce, reuse, recycle the object instances you use
- Strings, don't do big string concatenations using "+", use StringBuffer class instead
- Image, small is good?split large images, create a Image cache
- Object pooling might work in some cases
- Loops, avoid unnecessary creation/disposal of variables inside loops
- Use switch-case instead of if-blocks, they are translated to faster java bytecodes
- Use public variables directly instead of using get/set methods.
- Set variables to null when you don't need them anymore
- Garbage Collector, call frequently and explicitly on some devices. Beware of non-compacting GC
- Use local variables instead of global variables when you can. Local variables are faster and use less bytecode

Take a look at the following example:

```
for(y = 0; y < oriHeight; y++) {
    for(x = 0; x < oriWidth / 2; x++) {
        int idx1 = y * oriWidth + x;
        int idx2 = (y + 1) * oriWidth ? 1 - x;
        curPixel = buf[idx1];
        buf[idx1] = buf[idx2];
        buf[idx2] = curPixel;
    }
}
```

If we do the index creation outside of the loops like this:

```
int idx1;
int idx2;
for(y = 0; y < oriHeight; y++) {
    idx1 = y * oriWidth;
    idx2 = (y + 1) * oriWidth ? 1;
    for(x = 0; x < oriWidth / 2; x++) {
        curPixel = buf[idx1];
        buf[idx1] = buf[idx2];
        buf[idx2] = curPixel;
        idx1++;
        idx2--;
    }
}
```

It can save a few seconds on a real device!

Size

The size restriction for Java ME applications comes from two sources:

- Device capability, some devices only accept 64 kb applications!!
- Operator's gateway limitation, to avoid customers waiting too long (and pay too much) for an application to download some operators limit the max application size (normally around 300kb)

Device JAR size limit has improved continuously over time, but it is never enough for developers! So here some tips to help you

- Minimize classes number, avoid OOP (if practical), each class/interface contribute at least 250 bytes of overhead after compression. Can get away with 2 classes: one MIDlet class and one Canvas class.
- Use an Obfuscator
- Optimize Images
- Change ZIP algorithm, JDK's JAR utility is not optimal. Use open source, freeware or commercial alternatives, where number of passes are configurable, but beware of device compatibility.

Obfuscator

What does it do? It's original purpose is to make reverse engineering very difficult by:

- Eliminate packages (i.e. always use default package)
- Rename method/field names
- Remove unused code

But it has the nice side effect of creating smaller class files size (and also slightly faster). Some of them also perform bytecode optimization. They can typically reduce file size by 30-50%. There are very good Open source obfuscators (Proguard, Retroguard) and many other commercial products.

Images

What are the problems?

- Not compressible
- Each PNG file contains more or less the same header and footer
- Flipped images, transformed images on MIDP 2.0 devices are either very slow or not working. Isn't supported in MIDP 1.0 devices. Must include flipped version of sprites at build time this can double or quadruple the size

Here are the solutions:

- Use PNG optimizer like OptiPNG or PNGCRUSH
- Alternatively, do not compress image content - larger PNG sizes before compression but smaller at the end
- Combine multiple PNGs into a single resource bundle or a large image. It becomes more compressible and we stripped off the header and footer

How_to_optimize_in_Java_ME

- Flipped images
- Dynamically flip images at run time
- Potentially a performance/JAR size tradeoff
- Reduce colour depth (if possible)

See [How to optimize PNGs to reduce JAR size](#)