Java Porting Strategies

This article discusses Java porting strategies.

28 Jun 2009

Overview

In the world of mobile, Java never manages to achieve its *write once, run everywhere* objective. Mobile phones differ wildly in capability.

- Screen sized vary, from 96x65 to 800x480 (or to 640x360, even if you discount Windows Mobile devices)
- Heap sizes vary from 200k to 10s of Mb
- Execution speed varies by a factor of 100 between the slowest and fastest devices (source: JBenchmark 2.0)
- Some devices have no limitation on JAR size; for other devices, the limit can be as low as 64k

Developers are forced to choose between supporting a limited subset of devices, developing a very simple application (lowest common denominator), or developing multiple versions.

Add to that:

- Differences in which set of APIs are supported
- Differences in API implementation (but still to the specification)
- Bugs in the API implementation!

Even supporting two devices with the same specification can be a problem.

Primary Objective

Note: Our primary objective in porting is that, with each additional version ("SKU") we produce, the cost per SKU drops.

Porting Strategies That I Don't Recommend

Multiple Source Trees

A surprisingly popular technique (used within some of the major game publishers) is to develop a single version of the application, for one device (or a small subset of devices), then make dozens of copies of the source code. Each copy will then be modified to suit a specific device.

Advantages:

- Each build can be tailored to the device specification at the same time as working around device "issues"
- Fixes made to suit one device cannot break the build for another device
- You can have as many developers on the project as you have builds, decreasing the project duration

One big disadvantage:

- Fixes made to suit one device cannot *help* the build for another device, making it hard to achieve economies of scale, and increasing the total project cost
  - Any bugs in the original code will have been duplicated dozens of times over, creating a huge amount of duplicated coding and testing effort, and threatening product quality
  - A developer who increases the portability of the code (for example, by replacing "120" with "screenWidth / 2") helps only himself, not his other team members
  - Late specification changes become *very* expensive to implement

Using Pre-processors

This *very* popular technique involves using conditional compilation techniques used in C and C++ to maintain a number of code variants in a single source code file.

Pre-processors can also provide a number of other facilities, such as macro expansion (a trick for "in-lining" a method call and increasing performance).

Advantages:

- Single source tree reduces the total amount of code to maintain; the quantity of source code does not increase proportionally to the number of builds, creating an economy of scale
- Tools are readily available
- A well-established technique in the C community

Disadvantages:

- Looks fine when you use this for two or three device variants, but can quickly get out of hand, and can result in completely unreadable code if you're supporting 30 or 40 variants
- Depending on which pre-processor you use, your source code may no longer be valid Java (until it has been processed) - this can prevent you from using many of the features of modern IDEs, such as syntax checking, auto-complete, code navigation and refactoring
- Keeping your code as legal Java usually involves commenting-out large sections, depending on which version you're building - this can make it hard to work out which code is which
- Switching versions usually means uncommenting the code you do want, and commenting-out the code you don't - this can mean you change every source file, just by building a different version, and can result in a version control nightmare
- Advanced pre-processor techniques, like macro expansion, come with a set of gotchas that can trip even fairly experienced C programmers - Java programmers might never have written C, and might not realize the pitfalls

Recommended Best Practices

It may be necessary for you to produce multiple versions of your game or application to cover the range of handset models you want. Some techniques can help you minimize the number of different versions you will need; other techniques will help you produce multiple versions without having multiple source-trees.

Write Maintainable Code!

Before we even think about clever tools that can help us, by far the biggest benefit comes just from being a good programmer. Porting is an exercise in software maintenance. It's cheaper if the code is maintainable.

- Use a consistent style. Ideally, use Sun's Code Conventions for the Java Programming Language
- Use sensible variable and method names, and comment (again, use Sun's conventions for these)
- Write structured code (and use structured exception handling)
- Don't use literal constants in code, use named constants (static final)

Write Screen-Size Independent Code

This is really an extension of the previous item, but one that deserves special mention.

Make use of all the methods the API provides for receiving the size of the screen (through the Canvas.sizeChanged() event), and for querying the size of images and fonts.

Picking up image sizes at runtime allows you to adapt to different screen sizes just by changing the artwork (for bigger or smaller images), without needing any code changes.

Position items on the screen relative to where they belong. If something needs to be at the bottom, position it relative to the bottom of the screen, not the top. Remember: many devices vary only slightly in screen height. A classic example is Motorola devices, which are often highly compatible with each other, but may use a screen height of 174 or 175 pixels (fullscreen Canvas). You don't want to have to produce two separate builds to meet this one tiny difference (with two lots of development, testing, certification, etc.)

If you're displaying text in a Canvas, write a word-wrapping method. Don't break the text into lines manually! Even two different handsets *of the same model* can have different internal font sizes (Nokia 7250i, for example).

Develop to the Lowest Specification

You need to make some decisions in advance about what range of devices you want to support. Use the manufacturer developer sites, or sites like JBenchmark.com, to check for heap sizes and performance figures. Try to develop for the device with the least memory and the slowest performance.

Avoid using a Nokia S60 as your primary development device. S60s have more performance and more memory than the vast majority of devices, and you can end up with an application that is nightmare to get working on slower, lower memory devices. It is easier to port from a lower-spec device to a higher-spec device.

Don't necessarily work on the "worst" phone. You want to focus on developing a *product*, not on working around a plethora of firmware bugs. Nokia, Sony Ericsson and Motorola devices generally make the best choices, as they are reasonably stable, have good developer support, and have high levels of compatibility with other devices from the same manufacturer.

Use the Java Verified table of supported devices to get an idea about device compatibility. Ideally, choose devices from the "lead device" column.

Beware Multi-Threading

Be careful about using many threads. Different devices are likely to use different thread scheduling algorithms, which can make multi-threaded code behave very differently across devices if you're not careful about synchronization. Remember that only one thread can actually run at a time, and Java does not have as many rules about how thread scheduling should work as you might think. Few phones have sophisticated multi-tasking, multi-threaded operating systems like your desktop computer has, so be careful not to expect your phone to work quite so smoothly.

Do use multi-threading to avoid performing lots of work in an event handler method. Event handlers should return as quickly as possible. Failure to return quickly may result in applications that stutter or become unresponsive.

Be careful about using synchronization in `Canvas.paint()`, especially if you are using `serviceRepaints()`. You can inadvertently cause deadlocks on some devices.

Other Inconsistent Behaviours

- Avoid game actions (`Canvas.getGameAction()`). Mappings between actual keys and game actions vary between devices. Often, multiple keys will map to the same action. For example, depending on device, the FIRE action might result from the direction pad (or joystick) centre, "5", "0", left soft-key or send (green key). Or some combination. This can cause problems if, for example, you wanted to use "0" for a different action. It can also make writing help text a problem, since you don't know which keys the user will need to use. (Java Verified requires that user controls be documented in the help.)
- Reading to a byte array from an `InputStream`. Always check the return value to see how many bytes were *actually* read.
- Converting characters to bytes (and vice versa). Beware of methods that use the "platform default encoding", such as `String(byte[])` or `String.getBytes()`. These will behave differently on different devices. Consider `getBytes("UTF-8")`.
- `Timer` and `TimerTask` can be problematic on some devices, I recommend avoiding them.
- LCDUI looks different on different devices. Remember that you have no idea where Commands will appear. They won't necessarily be assigned to a soft-key, as some devices have a special, dedicated "back" key, which might be used for BACK Commands.
- Some devices struggle with very large images. This is not just a matter of memory, but often of dimension. Some devices cannot handle images beyond a certain maximum width and/or height. For example, an image 4096 wide by 1 pixel tall might fail, even though the memory requirement is small, while an image 128x32 (same number of pixels) might load fine. As a rule of thumb, the maximum dimension of an image *in either direction* should be:
  - 256 pixels, for devices with screens less the 176 pixels wide
  - 1024 pixels, for devices with screens 176 pixels wide or wider
- `platformRequest()` will require applications to exit on some devices before it does anything. Check the return value.
- Incoming calls (and other external events) may result in a `pauseApp()` event. Or not. If you're displaying a Canvas, then you might get a `hideNotify()` event. Or not. On some devices, you will get no event at all. The VM might continue running, or it might freeze until the call is completed.
- `startApp()` might get called more than once. Be careful not to re-initialize variables the second time around.

For the last two, see How to Suspend Java Game on Interrupt.

Create a "Device Properties" Class

For properties of a device that cannot be read from the device at runtime, create a class to describe each device. (This class should *not* include the screen size!! You can get that at runtime!)

Create a generic device class, with *all* properties.

```
abstract class GenericDevice {
    // the number of sounds that the device can have in the prefetched state at once
    public static final int MAX_PREFETCHED_SOUNDS = 1;
    // true if the device suffers from heap fragmentation and needs regular System.gc()
    public static final boolean NEEDS_REGULAR_GC = false;

    // true if RMS access on this device is very slow (10 seconds or more)
    public static final boolean RMS_IS_SLOW = false;
}
```

A device-specific class extends this, and provides its own values as needed.

```
public class Device extends GenericDevice {
    public static final int MAX_PREFETCHED_SOUNDS = 3;
}
```

Properties should relate to specific, functional characteristics of devices. They should *not* refer to manufacturer, model, or operating system. "NOKIA_SERIES_40" is not a useful thing to know.

Create a separate Device class for each version you will produce. Sharing a single `GenericDevice` class makes it easy to add a new property, with a default value, without having to edit every single Device class.

Use Conditional Compilation

You can do this in Java without the aid of a pre-processor.

```
if (Device.NEEDS_REGULAR_GC) {
    System.gc();
}
```

Since `Device.NEEDS_REGULAR_GC` is a static final, it has a primitive type, and it is initialized from a constant, its value is known to the compiler at compile-time. If the value is "true", the compiler will ignore the "if", and leave just the `System.gc()`. If the value is false, the compiler will ignore this entire code fragment.

Use Abstraction

Cope with different device APIs by creating your own abstraction layer. This is a carefully-targeted variation on using multiple source trees. Essentially, it is a *device driver pattern*.

Two examples:

1. If you need to use different sound player APIs (or even different implementations of JSR135!), create your own sound-player interface, then device-specific implementations.

   ```
   public interface SoundPlayer {
       public void loadSound(String name);
       public void setLooping(boolean looping);
       public void start();
       public void stop();
   }
   ```

   You'll only have one class in your JAR that implements the interface, and newer versions of Proguard are able to detect this and eliminate the interface from the build.

2. If you need to extend `Canvas`, `GameCanvas` or `FullCanvas` depending on device, create an intermediate class (for which you will have different, device-specific versions).

   So, instead of this:

   ```
   public class MyCanvas extends
   //#if MIDP2
       GameCanvas
   ```

```
//#elseif NOKIA
    com.nokia.mid.ui.FullCanvas
//#else
    Canvas
//#endif
{
```

You just have:

```
public class MyCanvas extends DeviceSpecificCanvas {
```

A `DeviceSpecificCanvas` class looks something like:

```
public class DeviceSpecificCanvas extends Canvas {
    // might not even need any code
}
```

There is a per-class overhead in a JAR of around 150 bytes per class. However, tools like JAX, mBooster and newer versions of Proguard can merge classes. `DeviceSpecificCanvas` and `MyCanvas` (in the example above) can be merged without any risk to the functionality of the code.

Re-Use

Re-use code that has already been through a porting cycle, so you don't have to port it again. If you're working in a sensible way, each time around the cycle it will become more portable, and have fewer bugs.

Re-use is obviously easiest if you can exploit object orientation. If you can't, because JAR size constraints stop you having more than a few classes, then you can consider the "class stacking" technique.

The Class Stacking Technique

Not a best practice, but I've used the term and it warrants explanation.

As mentioned before, there are various tools (such as JAX and mBooster) that can merge classes. This works best when:

- One is the super class of the other
- The super class is abstract (or is, at least, never instantiated)
- They do not have identically named, non-private fields or methods

If these are met, then the classes can be merged into a single class, without increasing the amount of heap required to hold an instance.

It is possible, then, to build a tall, thin class hierarchy, one class wide and up to 20 or 30 classes tall (I don't recommend making it taller than 30 classes). The process of class merging can then reduce them all down to a single, large class.

In effect, this mimmics modular programming and static linking, as is common with C programs.

As I say, this is not a *best practice*. However, it is better than trying to squeeze all your code into one huge class (or, as I've seen on at least one occasion, one large paint() method and one large run() method). It does at least provide for some re-use, API abstraction, and helps increase the number of developers who can work on the project (without having to merge changes all the time).

Use Byte-Code Engineering

BCE is a technique for modifying the program in an automated way, *after* compilation. That is, the .class files themselves are modified, usually by *injecting* additional byte code. If you hear people referring to "AOP" (Aspect Oriented Programming), this is usually what they are talking about.

BCE is ideal for coping with bugs in API implementations, by enabling you to inject a standard fix. Because there are no source code changes, the source code remains readable, and you don't risk breaking the code for other devices.

Fixes for common, known issues can be encapsulated in re-usable components, so that you never have to think about fixing that issue again.

This technology has been the basis of porting tools such as Tira Wireless's "Jump" product (a commercial product, which is now, to my knowledge, defunct), and also of the coincidentally similarly-named, open source project "GUMP".

For example, at least one device that returns an incorrect value from `Canvas.getHeight()` when in fullscreen mode (it returns the non-full-screen height). You need to replace any call to this method with the correct value, and you need to do this for every game you develop on that device. This fix can be simply a matter of:

```
replaceCalls(allClasses(),
        // this is the signature for the calls we want to replace
        "javax.microedition.lcdui.Canvas.getHeight()I",
        // this represents the code we're going to inject to replace each method call
        "{ $_ = 160; }"
);
```

Here, "$_" is a special place-holder to represent the return value from the call we're replacing. The modified code will function exactly as if `getHeight()` had been called, and 160 had been returned.

Note that this code does not go in the game or application. It is executed by the BCE tool as part of the build process, modifying the code in the JAR. If using GUMP, for example, this becomes part of a library of "gumplets", each of which fixes a specific issue.

In Conclusion

Porting to the largest number of devices at the lowest possible cost is obviously key to maximizing the return on your investment - in the mobile Java world at least.

There are many tools and techniques that can help you. There are commercial products, and there are free products. Some of these are useful, and can help you significantly. None of them is an alternative to good software engineering.