

# Using Java in embedded systems

By Brian J Murphy

# Introduction

- Senior software engineer with APC with responsibility for touch screen displays on large 3 phase UPS's (Uninterruptible Power Supplies)
- BSc. In Applied Physics and Electronics from NUI, Galway.  
MSc. In Computing from GMIT
- Working with embedded Java for about 6 years
- This presentation is going to give a high level summary of what I have learned about embedded Java.

# Agenda

- Why use embedded Java?
- Drawbacks of using Java in embedded applications
- Examples of embedded JVM's and tips for choosing the correct one
- Typical design considerations
- Things to watch out for during implementation
- Real time and Java

# Why use Java in embedded applications

- The ability to write the core of an application once and run it anywhere.
- Potential for greater reuse across different platforms
- OS independence
- Vendor neutrality
- Platform neutrality
- Access to a huge array of libraries
- Standard API's
- Access to highly productive development environments
- Large development community
- The ability to scale applications easily using off the shelf components
- Dynamic memory management

# Drawbacks to Java in embedded systems

- No direct memory access and no interrupt handling
- Can use more memory. VM's have different strategies for this but typically more memory is used than that of a C/C++ application
- You still have to cater for hardware and protocols the JVM does not know about such as CAN, RS485 or GPIO
- Lots of different standards for embedded VM's. MHP, J2ME CDC etc. Can be confusing
- Dynamic memory management. The GC task can pre-empt threads and cause delays on your system if you are not careful

# Performance

- The first generation of JRE's were 10 to 20 times slower than the native application.
- Current JRE's/JIT's can run most applications as fast as C
- Some chip manufacturers provide hardware acceleration (ST, Amdel) that will execute 70% of Java op codes directly on the CPU, while referring opt codes the CPU doesn't know about back to the JRE (JRE must support this feature)
- Badly written code in any language runs badly everywhere and Java is no exception, but, its ease of use gives an inexperienced developer enough rope to really hang themselves. You may get away with it on a Quad core X86 with 16Gb of memory but not on a resource constrained platform.
- Beware of early articles on Java performance. They usually do not reflect the current state of technology.

# Compilers

- Standard way is interpretation
- JIT is impractical for embedded systems. Too much memory required

	Dynamic adaptive compilation	Ahead of time compilation
Advantages	<ul style="list-style-type: none"><li>•Resident &amp; downloaded code</li></ul>	<ul style="list-style-type: none"><li>•Global optimization</li><li>•Fastest execution speed</li></ul>
Disadvantages	<ul style="list-style-type: none"><li>•Additional RAM requirements</li><li>•Limited cache of instructions</li></ul>	<ul style="list-style-type: none"><li>•Resident code only</li><li>•Additional ROM requirements</li><li>•Profiling requires</li></ul>

- We never had a need to do any of this.

# Selecting a VM

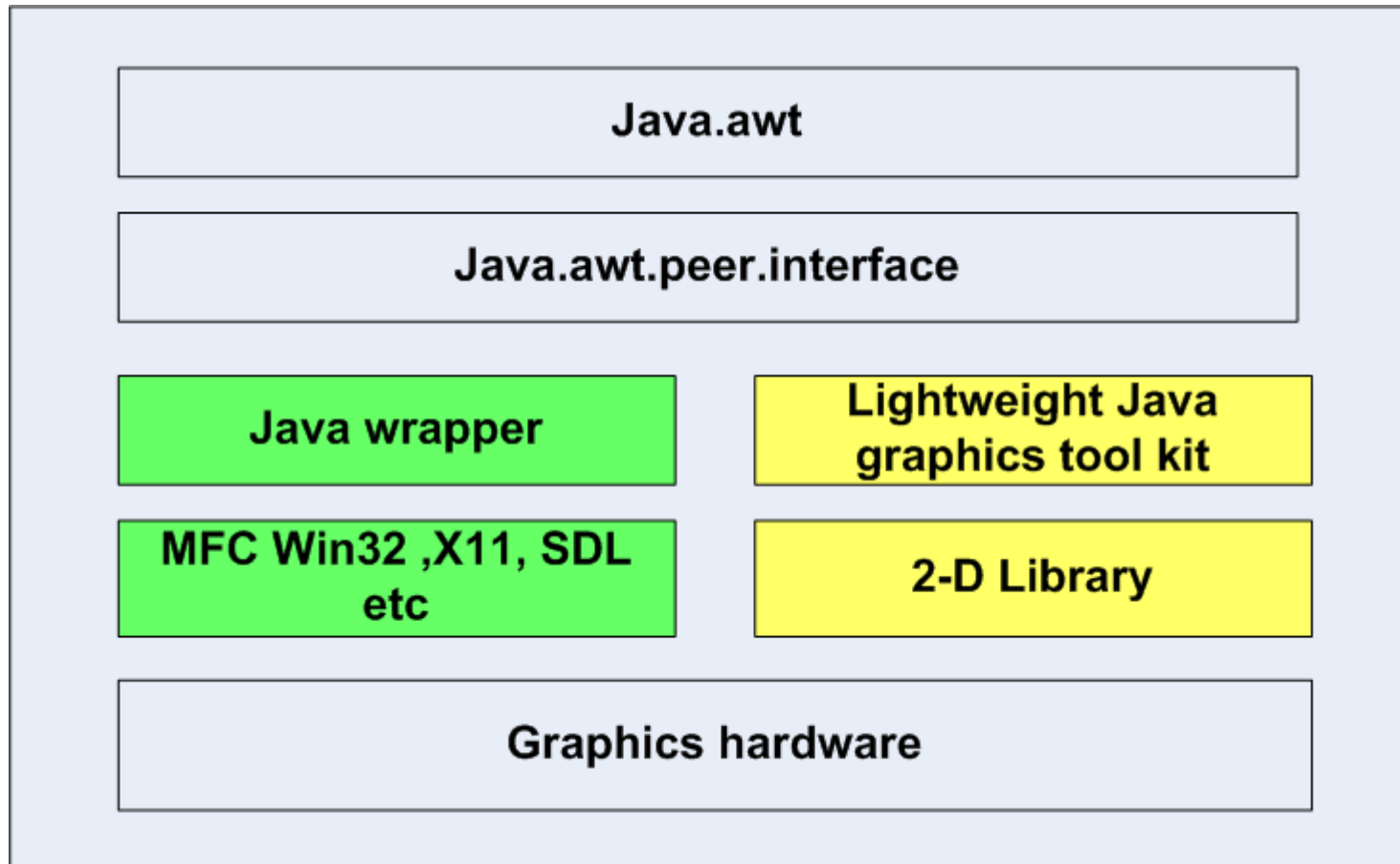
- How many platforms does the VM run on? Swapping VM's can be troublesome so a VM that runs on lots of platforms is good.
- What are you willing to pay? Open source is good but make sure your VM has been used for a similar application. This also holds true for commercial ones.
- What OS? Linux is usually the preferred OS due to the fact that it runs on so many platforms and is supported by so many vendors now. You can also interact with hardware a lot easier due to the UNIX Principle that everything is a file.
- What type of functionality do you need? There are lots of standards supported for different industries. MHP(multi media), CDC (Mobile phones). CLDC (PDA's consumer device)

# Selecting a JVM

- The real limiting factor is footprint. The more space you have the more you can fit in. We have deployed embedded java applications from 700K using JamVM up to 120Mb using the full Java SE
- You should look carefully at what classes the VM provides natively. If you require additional classes you must add them to your application and this can increase your footprint.
- If you are implementing a graphical application you need to look carefully at how your VM integrates with your hardware. Some VM's can talk directly to the frame buffer while others require an xserver, SDL
- You also need to find out what additional native libraries the VM may need. For example, on an ARM processor you may need a library for floating points
- Find out if there are any special requirements needed by the VM

# Selecting a VM

*Typical structure of and embedded VM*



# Typical system requirements

	Minimum requirements	Medium requirements	High-end requirements
Processor	<ul style="list-style-type: none"> <li>•50 to 100 Dhrystone MIPS</li> <li>•&lt;100Mhz clock</li> <li>•No FPU</li> </ul>	<ul style="list-style-type: none"> <li>•150 to 250 Dhrystones MIPS</li> <li>•100 to 200Mhz clock</li> <li>•FPU Optional</li> </ul>	<ul style="list-style-type: none"> <li>•250 to 400+ Dhrystone MIPS</li> <li>•250+Mhz clock</li> <li>•Hardware FPU</li> </ul>
Memory	<ul style="list-style-type: none"> <li>•2Mb Storage</li> <li>•2 Mb RAM</li> </ul>	<ul style="list-style-type: none"> <li>•2 to 4+ MB storage</li> <li>•4 to 8MB RAM</li> </ul>	<ul style="list-style-type: none"> <li>•&gt;8MB Storage</li> <li>•&gt;16Mb RAM</li> </ul>
Java Apps	<ul style="list-style-type: none"> <li>•Typical non graphical. Text based</li> </ul>	<ul style="list-style-type: none"> <li>•Moderately graphical</li> </ul>	<ul style="list-style-type: none"> <li>•Intensively graphical</li> </ul>

# The JVM

- There are many JVM's out there. A good list is provided at <http://www.kaffe.org/links.shtml> We will take a quick look at a few of them, just as an overview.
- NanoVM (Open Source) which runs on a 8 bit AVR Mega 8 and has a footprint of 8kbytes and gives you
  - java/lang/Object (object handling)
  - java/lang/System (IO handling)
  - java/io/PrintStream (console output)
  - java/lang/StringBuffer (string processing)
  - asuro (asuro control)
  - Native support for LCDs, AVR io etc
  - <http://www.harbaum.org/till/nanovm/index.shtml>

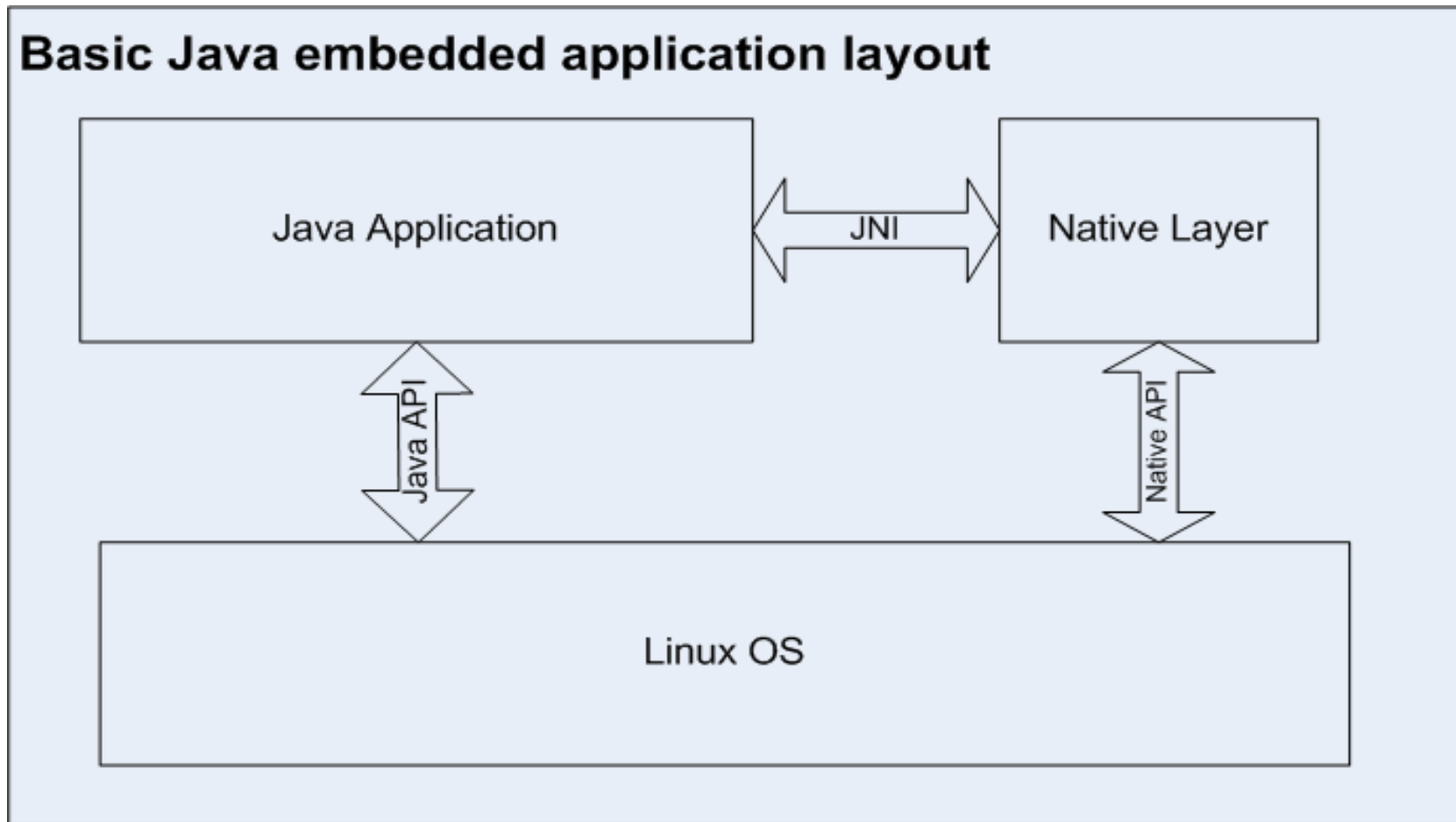
# The JVM continued

- CEEJ VM (commercial Skelmir [www.skelmir.com](http://www.skelmir.com))
  - Clean-room implementation of Java
  - Platforms PPC x86, MIPS 32, MIPS 64, ST20, ST40, ARM, StrongARM, Xscale, SH2, SH3 & SH4 Equator BSP-15, Equator MAP-CA, TI OMAP & DM642 DSPs & VLIWs SPARC
  - Microsoft Windows 98/NT/2000/XP, GNU Linux, pSOS, WindRiver VxWorks, Apple MacOS/X, Microsoft Windows CE, Microsoft PocketPC, Sun Solaris, Sun Chorus, ATI Nucleus RTOS
  - Set-top boxes, Digital Televisions, Multi-Function Office Machines, Handheld wireless communication, Residential gateways & modems, Industrial applications, Kiosks, Game consoles.
  - Full J2ME personnel profile in 1.8 Mb that includes AWT.

# The JVM continued

- Sun JVM
- Java SE embedded
  - For high end embedded systems. Basically, it allows you to pick and optimize SE for your embedded application with some additional tools.
- J2ME
  - The Java ME CLDC and CDC offerings are designed for resource-limited devices including cell phones, handsets and media players. The API's of CLDC and CDC are more limited than Java SE but enable these ME offerings to support devices with small footprints.

# Design Considerations



# Design considerations

- JNI

- Don't be fooled. Lots of vendors will offer what they claim are quicker and easier to use API's but you will tie yourself to that vendor's API and will gain little from it in the long run.
- Invocation API makes your application a lot easier and neater to control. It's easier to port and allows a more seamless integration with the native layer
- You can do most things in Java but sometimes it makes more sense to do things natively. Also, you may have customized hardware that needs to be controlled.
- Think carefully about where you place functionality. The more you place in the native layer the more difficult it is to port. Typically we try to keep the native layer as thin as possible. OS calls should be wrapped to aid portability.

# Implementation details

- Native threads should never do work in Java and Java threads should never do work in the native layer.
- Avoid using inner classes and anonymous classes. Each inner class, no matter how small, will create a class file 4K in size. There are tools that can reduce this but avoid them if you can.
- When invoking your VM set the memory footprint using `-Xms` `-Xmn`. These should have the same value. This will cause all the memory for the VM to be allocated up front and is similar to overriding `malloc` in c.
- Use the `new` key word sparingly. Recycle objects. This reduces the load on the garbage collectors.
- Know your garbage collector. There are several different approaches used for GC. Some VM's allow you to select the type of algorithm to use

# Implementation details.

- It is possible and many well designed applications can run without a GC.
- From experience, the GC only becomes an issue when you do something stupid in your code.
- Profile the application to ensure it is behaving as you expect.
- Use an application like GenJar or manually include the class files you need to deploy. You can do this manually by setting the jre to verbose mode, but, this is time consuming.

# Java Real time

- Defined by JSR001 in the Java community process.
- Timesys and Sun have reference implementations.
- The RTSJ addresses the critical issues by mandating a minimum specification for the threading model (and allowing other models to be plugged into the VM) and by providing for areas of memory that are not subject to garbage collection, along with threads that cannot be pre-empted by the garbage collector.

# Conclusions

- Java is another tool in the embedded developers toolbox
- It will not be suitable for every project
- Performance is generally not an issue on the larger embedded systems
- Care needs to be taken with managing resources - it is the same as with any embedded application
- It is very flexible and has oceans of libraries that can be used