Tantalum5 Quick Start Manual

How to get started with asynchronous concurrent functional programming, the promises pattern, fork-join-chain and advanced web service caching to create online-offline services for JME, Android and cross-platform. We cover basic worker threads, examples of cross-platform application logic and simplified threading. Be sure to also check the extensive javadocs for details.

Paul Houghton



Contents

[Introduction 2](#_Toc352020948)

[The Tantalum Project Team 2](#_Toc352020949)

[License 3](#_Toc352020950)

[Platform Support 3](#_Toc352020951)

[Source Code and Examples 3](#_Toc352020952)

[Tantalum Overview 4](#_Toc352020953)

[Initializing and Closing Tantalum 4](#_Toc352020954)

[Asynchronous Worker Thread Model 5](#_Toc352020955)

[Networking 6](#_Toc352020956)

[Simplified XML Parsing 6](#_Toc352020957)

[Simplified JSON Parsing 8](#_Toc352020958)

[Cross-platform Implementation of Android’s AsyncTask 10](#_Toc352020959)

[Fork-Join Pattern Adapted for Rich UI Threading 10](#_Toc352020960)

[Caching 11](#_Toc352020961)

[Setting Up Your First Project 12](#_Toc352020962)

[Creating Multiple Build Configurations 12](#_Toc352020963)

[Setting Up the Android Project 13](#_Toc352020964)

[Separation of Application Logic and User Interface 13](#_Toc352020965)

[A Word on Obfuscation 14](#_Toc352020966)

[Summary 14](#_Toc352020967)

[About the Author 15](#_Toc352020968)

# Introduction

In November of 2010 Paul Houghton was teaching a motivated group of developers in Pretoria, South Africa. Part of that included opinions on how mobile Java applications should be created and structured to create a great user experience (UX), always fast even in poor network conditions. The understandable response was “OK, show us”. The challenge was accepted, and at 4AM the following morning, and only 1 error later, Tantalum 1 was born as a library for simplified networking, parsing, persistent data caching and supporting high performance user interfaces with concurrency.

Tantalum has come a long way since the first release crystallized a “small and fast is good” view of concurrent and mobile Java application development. Tantalum 5 is the result of many hands and new ideas. It has grown into a cross-platform Java framework that produces a single JAR that runs on both Android and JME. It recently became clear that Tantalum was very, very similar to other asynchronous functional programming (FP) paradigms such as C++ 11 futures and JavaScript promises. We never set out to follow a trendy buzzword. We just want to make it easy to create reliable and fast programs with least time and effort. But we find we are FP, so we started to use that body of knowledge to further simplify and speed up our code. And since FP is like Tantalum, it must be good.

Tantalum has been used very successfully in a number of commercial applications delivered by Nokia, Futurice and others. New projects are coming up all the time. Nokia is strongly embracing Tantalum and adopting it as an official part of their Asha SDK. Now Android and Nokia developers no longer have to choose- the combined market is 3x the size of Android alone and the tools are fun to use.

We started by focusing on UX and speed. Tantalum development also succeeds because it makes you more productive and gently structures your application for real world performance. Tantalum 5 expands on this legacy to bring additional capabilities for:

* caching of web services for speed, cost-savings and online-offline use
* simple fork-join-chain thread pool combination and sequencing of Task objects. The result is simple code that never blocks the UI thread while gives UX speed through concurrent, asynchronous IO.
* platform-independent business logic to save time and expand your application’s market footprint
* FP principles and clean design best practices are introduced the easy way, without heavy theory or having to learn a new language

You can find the latest full source and other news for the project at <https://github.com/TantalumMobile> . The latest pre-packaged download is available from <https://projects.developer.nokia.com/Tantalum> .

## The Tantalum Project Team

Tantalum is the combined work of many people. Paul Houghton started the project based on long experience making really, really fast mobile applications that are a pleasure to use. Significant contributions by Kai Inkinen, Kingsley Adio, Timo Saarinen, Oskar Ehnström, Samuli Saarinen, Vera Andersson, Mark Voit, Kai Inkinen, Satyam Banderapu, Ville Vainio and many others have been invaluable. A special thanks to Nokia and Futurice for their generous and (mostly) willing sponsorship of this work through its infancy. We look forward to Nokia taking it forward to the next level.

This is open source. Small box development is great. We are a community of skilled developers playing around and making great stuff for several platforms at the same time. We welcome you to join us at <https://github.com/TantalumMobile/> .

## License

Tantalum has been adopted by Nokia as an official part of their developer offering. It is currently in use in commercial software used by millions and licensed under a permissive BSD open source license, <http://opensource.org/licenses/BSD-2-Clause> . This means you can freely include the code, as either a library or modified or unmodified source code, in your project. You must acknowledge use of the code in your documentation and “About”. You are not obliged to publish your modified source code, but please do provide your suggested improvements as this is open source. Normal, minimally-restrictive legal stuff. The license text is visible at the top of the source code.

## Platform Support

Note that although the examples here refer to and use the Nokia Software Development Kit (SDK), there are no dependencies on Nokia proprietary libraries in Tantalum. You may equally use Tantalum with other JME platforms such as LG and Samsung phones including Bada. The Android package does depend on Android and J2SE packages, and you can find these in a light adaptation class called PlatformUtilities. Other platform support will be added based on community interest. There is a rumor of a Blackberry port, and other Java variants are a natural step.

## Source Code and Examples

You can find everything at Tantalum Project, <https://github.com/TantalumMobile/> . This includes a BBC RSS Reader example application implemented using three different user interfaces: JME Forms, LWUIT, and JME Canvas using Nokia’s proprietary Series40 extensions. It also includes simple web service examples for converting a plain text address into latitude and longitude, with and without caching the results.

The best cross platform example using Tantalum 5 is available at <http://projects.developer.nokia.com/picasa_viewer> . This demonstrates a Canvas-based UI that runs on Nokia SDK 1.0, 1.1 and 2.0 phones and an equivalent UI on Android. A nice development shortcut to learn this approach to cross-platform development is to start with and test this example, then change the data model and UI to suit your application needs.

You may choose to include a pre-build JAR into your project. You can more easily step into the code if you pull the full source code of the library as a project, and refer to this project from your own project. This approach also lets you more easily submit your improvements back to the Tantalum community for all to benefit. That is how it should be done.

To use the latest development versions, you should install git on your computer and then open a command prompt in the directory below which you want Tantalum to appear and type:

**git clone git://github.com/TantalumMobile/Tantalum.git**

# Tantalum Overview

Let’s take a quick run through what Tantalum offers and some code samples of these features in use. To keep to the Tantalum fast and light approach, we will not dwell on each feature but rather illustrate it. More complete information is available in the API documentation, the source code itself, and demonstrated in the complete working examples.

## Initializing and Closing Tantalum

You only need one line of code to add Tantalum to your JME MIDlet or Android Activity. Here we specify the main program, the thread pool size, and for JME one of three logging modes.

(JME)  
public MyMIDlet() {  
 PlatformUtils.getInstance().setProgram(this, 4, JMELog.MEMORY\_CARD\_MODE);  
}  
  
(Android)  
public MyActivity() {  
 PlatformUtils.getInstance().setProgram(this, 6);  
}

Closing the application on application exit gives a clear write and close of resources in use. You must specify if you want the calling thread to be help

(JME)  
public MyMIDlet() {  
 PlatformUtils.getInstance().setProgram(this, 4, JMELog.MEMORY\_CARD\_MODE);  
}  
  
(Android)  
public MyActivity() {  
 PlatformUtils.getInstance().setProgram(this, 6);  
}

You only need one line of code to add Tantalum to your JME MIDlet or Android Activity. Here we specify the

In J2ME, the simplest initialization is to let your MIDlet extend the TantalumMIDlet class. On Android your main Activity should extend the TantalumActivity class. This links the library into the application lifecycle on each platform. The worker threads and logging facilities will be automatically initialized for you, and ongoing operations such as flash memory read and write will be completed during app exit.

You can close your program cleanly on both platforms at any time by calling Worker.shutdown(boolean unconditional). If you set unconditional false, the shutdown tasks can take as long as necessary to including completing current tasks. With unconditional set to true, the background tasks will be interrupted after 3 seconds and they may not exit cleanly.

## Asynchronous Worker Thread Model

The Worker class maintains a queue of background operations to be completed on one of several Worker threads. You can queue a piece of code for background execution. Think of this as a lambda expression if you are familiar with the concept. Lambda expressions will be available in Java 8, but they are not currently available on any mobile Java platform.

Queuing a piece of code, or “forking” it to another thread for execution, is done by calling Task.fork() where priority is usually one of the following most common types:

1. Task.NORMAL\_PRIORITY – The task is added to the end of the queue and execution will start after all previously queued tasks. This is the default operation if you fork.
2. Task.HIGH\_PRIORITY – The task is added to the beginning of the queue and execution by the next available Worker thread will begin soon unless subsequent HIGH\_PRIORITY tasks jump in front of this task on the queue. This is useful for user interface code where the task will result in an operation visible to the user on the current screen.
3. Task.FASTLANE\_PRIORITY – The task will be executed by the next available thread, including one fastlane worker thread reserved for only fastlane tasks. Tasks in the fastlane generally update the user interface and should not take more than a few milliseconds to complete.

Additional priorities DEDICATED\_THREAD\_PRIORITY, UI\_PRIORITY, SERIAL\_CURRENT\_THREAD\_PRIORITY, SERIAL\_PRIORITY, IDLE\_PRIORITY and SHUTDOWN are also available. Check the JavaDoc for details and recommended usage.

Most operations are from a single common task queue with multiple treads pulling tasks from the queue. Execution order is in this case mostly predictable but not guaranteed. Task.fork() can be used to guarantee sequential execution by a single Worker threads. SHUTDOWN priority can be used to add work such as closing resources that will be completed automatically before the program exits.

The base class threading work is Task. Each task holds its current value and will complete a single call to exec() using the thread and sequence specified by its priority. These actions are clearly displayed in the application log to make it easy to see which actions are on background threads, the UI thread etc. Each Task is extended by your code to implement exec(). An alternate usage pattern discussed in the fork-join-chain section below. You can for example force out-of-order execution and convert asynchronous methods to synchronous by calling Task.join() for the completed result.

Each Task has an internal state which starts as Task.PENDING. This changes automatically after the task is queued and executed or if it cancel() is called before it completes exec(). The other states are:

* Task.FINISHED – The Object exec(Object in) method inherited from Workable complete normally on a background Worker thread
* Task.CANCELED – An exception was thrown or there was an explicit call to cancel()before the execution completed. The Task’s onCancelled() method will be called on the UI thread

Because a Task manages the state changes for you and protects you from errors like forking the same task several times, you cannot override the exec() directly as in Workable. Task will handle the state changes- simply include your code in an the exec() method.

new Task(Task.NORMAL\_PRIORITY) {  
 public Object exec(Object in) {  
 // Do something on a background Worker thread  
 }  
}.fork();

You may have noticed that each Task receives an input and returns an output result. This is used to allow the automatic and easy chaining of asynchronous tasks one to the next. Execution of the next step in the chain often proceeds on a different thread. Each Task can be run only one time, it becomes a functional programming immutable final value object after execution completes.

## Networking

All this threading and Task is interesting, but what can we use it for? HttpGetter and HttpPoster are Tasks that support the network HTTP\_GET and HTTP\_POST operations asynchronously. Exception handling, cancellation and chainable results are returned in the same way as other Task objects you may implement. Code can be passed to the class and executed automatically after the GET or POST operation, and you can optionally include alternate code to run if the network operation fails as happens in real world mobile networks. You may specify how many automatic-retries will take place if there is a network error before the operation is cancelled.

HttpGetter httpGetter = new HttpGetter(Task.HIGH\_PRIORITY, url).chain(new Task(Task.FASTLANE\_PRIORITY) {  
 public Object exec(final Object in) {  
 // Perform an operation on LOR “in” (a pointer to the byte  
 // array) received from the net.  
 // This is called from a background worker thread  
 return in;  
 }  
 protected void onCancelled() {  
 // On the UI thread, update the screen when the operation fails  
 };  
}  
httpGetter.fork(); // Queue for worker thread background execution

Notice that we have very fine control over the sequence and priority of task execution, and each step in the execution can have a different set of threads it may run on or even it’s own dedicated thread.

## Simplified XML Parsing

SAX parsing of XML with the default J2ME JSR is rather tedious. It does not give information about the context, only a string of tags which must be deciphered. To make this easier, XMLModel maintains a stack of tags and other housekeeping to make parsing XML into a Java object relatively simple. To illustrate use, see how XMLModel is extended into RSSModel to create a Vector of RSSItem objects.

public class RSSItem {  
 private String title = "";  
 private String truncatedTitle = null;  
 private String description = "";  
 private String link = "";  
 private String pubDate = "";  
 private String thumbnail = "";  
 private volatile boolean loadingImage = false;  
 private volatile boolean newItem = true;  
 private Font truncatedFont;  
 private int truncatedTitleWidth = 0;

public synchronized String getDescription() {  
 return description;  
 }

public synchronized void setDescription(String description) {  
 this.description = description;  
 }

.. // Similar getters and setters for each field  
}

The parsing in RSSModel is then a series of startElement() – parseElement() – endElement() calls which build up the Vector of RSSItem objects.

public class RSSModel extends XMLModel {  
 protected final Vector items = new Vector(40);  
 protected RSSItem currentItem;

..

public synchronized void startElement(final String uri, final String localName, final String qName, final Attributes attributes) throws SAXException {  
 super.startElement(uri, localName, qName, attributes);  
 if (qName.equals("item")) {  
 currentItem = new RSSItem();  
 }  
 }

protected synchronized void parseElement(final String qname, final String chars, final XMLAttributes attributes) {  
 try {  
 if (currentItem != null) {  
 synchronized (currentItem) {  
 if (qname.equals("title")) {  
 currentItem.setTitle(chars);  
 } else if (qname.equals("description")) {  
 currentItem.setDescription(chars);  
 } else if (qname.equals("link")) {  
 currentItem.setLink(chars);  
 } else if (qname.equals("pubDate")) {  
 currentItem.setPubDate(chars);  
 } else if (qname.equals("media:thumbnail")) {  
 currentItem.setThumbnail((String) attributes.getValue("url"));  
 }  
 }  
 }  
 } catch (Exception e) {  
 //#debug  
 L.e("RSS parsing error", "qname=" + qname + " - chars=" + chars, e);  
 }  
 }

public void endElement(final String uri, final String localName, final String qname) throws SAXException {  
 super.endElement(uri, localName, qname);  
 if (qname.equals("item")) {  
 if (items.size() < maxLength) {  
 items.addElement(currentItem);  
 }  
 currentItem = null;  
 }  
 }

## Simplified JSON Parsing

A JSON parser for J2ME is included in Tantalum. This was not created by the Tantalum project, but simplification of conversion from text format into a fast and low-memory Java object has been added on top of the parser.

If we take the example of JSON data from the Picassa web service, we want to convert that into a Vector of PicasaImageObject objects.

public final class PicasaImageObject {  
 public final String title;  
 public final String author;  
 public final String thumbUrl;  
 public final String imageUrl;  
 public PicasaImageObject(final String title, final String photographer, final String thumbUrl, final String imgUrl) {  
 this.title = title;  
 this.author = photographer;  
 this.thumbUrl = thumbUrl;  
 this.imageUrl = imgUrl;  
 }  
}

We can then parse the JSON string as follows:

final Vector vector = new Vector();  
try {  
 o = new JSONObject(new String(bytes));  
} catch (JSONException ex) {  
 //#debug  
 L.e("bytes are not a JSON object", featURL, ex);  
 return null;  
}  
if (o != null) {  
 JSONArray entries = new JSONArray();  
 try {  
 final JSONObject feed = ((JSONObject) o).getJSONObject("feed");  
 entries = feed.getJSONArray("entry");  
 } catch (JSONException e) {  
 vector.addElement(new PicasaImageObject("No Results", "", "", ""));  
 //#debug  
 L.e("JSON no result", featURL, e);  
}  
for (int i = 0; i < entries.length(); i++) {  
 try {  
 final JSONObject m = entries.getJSONObject(i);  
 final String title = m.getJSONObject("title").getString("$t");  
 final String author = m.getJSONArray("author").getJSONObject(0).getJSONObject("name").getString("$t");  
 final String thumbUrl = m.getJSONObject("media$group").getJSONArray("media$thumbnail").getJSONObject(0).getString("url");  
 final String imageUrl = m.getJSONObject("media$group").getJSONArray("media$content").getJSONObject(0).getString("url");  
 vector.addElement(new PicasaImageObject(title, author, thumbUrl, imageUrl));  
 } catch (JSONException e) {  
 //#debug  
 L.e("JSON item parse error", featURL, e);  
 }  
}  
if (entries.length() == 0) {  
 vector.addElement(new PicasaImageObject("No Results", "", "", ""));  
}

## Cross-platform Implementation of Android’s AsyncTask

If you are already familiar with Android’s asynchronous UI approach, a full implementation is available with the AsyncTask extension of the Task class. This is generally the same stateful approach at Task, but has the added benefit that you can easily publishProgress() to the UI thread for a long running operation where the user should be informed that it is still progressing.

## Fork-Join-Chain Pattern Adapted for Responsive User Interface Design

Asynchronous UI code can sometimes be made simpler by writing it in a traditional, synchronous fashion. This is the approach used by Window 8 and Windows Phone 8 with the C# async and await keywords prepended to methods. We duplicate the usability of this pattern in Java and Tantalum by modifying and simplifying the scalable Java server fork-join framework of Java 7 to use it for patterns between the UI thread

Consider the following example taken from a painter running on the UI thread. We want to load data asynchronously and use that immediately if possible, but we do not want to make our UI unresponsive when we bog down the UI thread with slow network requests for an image. The solution is to join() the asynchronous request for a maximum of 100ms and if there is no result let the UI repaint when the image does arrive.

startSpinner(); // Display a spinner while the image loads  
try {  
 PicasaStorage.imageCache.get(selectedImage.imageUrl,  
 new Task(Task.HIGH\_PRIORTY) {  
 public Object exec(final Object in) {  
 if (in != null && selectedImage ==  
 PicasaStorage.getSelectedImage()) {  
 image = (Image) in;  
 stopSpinner(); // Hide the image load spinner  
 }  
 return in;  
 }  
 }).**join(100)**; // Wait max 100ms for result  
} catch (TimeoutException ex) {  
 // Normal for slow load if must HTTP\_GET over the network  
} catch (Exception ex) {  
 L.e("Can not join image load", selectedImage.imageUrl, ex);  
}  
// Draw the image if it is available after join(100) above

## Caching

StaticWebCache automatically fetches and converts your data in the following order:

1. **RAM Cache** – A WeakReference Hashtable that stores all of your data in RAM memory. When the Virtual Machine (VM) is low on memory, it will remove items at random from this cache. Items in the RAM Cache are stored as Java Objects such as Image objects (not byte arrays), and this can be referred to as “use form” as this is the form in which they can be readily and rapidly consumed.
2. **Flash Memory Cache** – Data which is not found in the RAM Cache is searched for in persistent storage. The implementation uses the Record Management System (RMS) on J2ME and an SQLite database on Android.
3. **Network Web Service** – Data which is not found in the Flash Memory is fetched from the web using HTTP\_GET. It will automatically be added to RAM and Flash Memory caches when it is received. Usually the Task which fetches the data will also put that data on the UI and possibly process it further according to the logic of the application.

You create a StaticWebCache for each type of data you with to handle. Each cache has a CacheView associated with it, and this will automatically convert from the HTTP\_GET result byte[] format into use form. In the example below, the CacheView parses the XML data in an RSS feed.

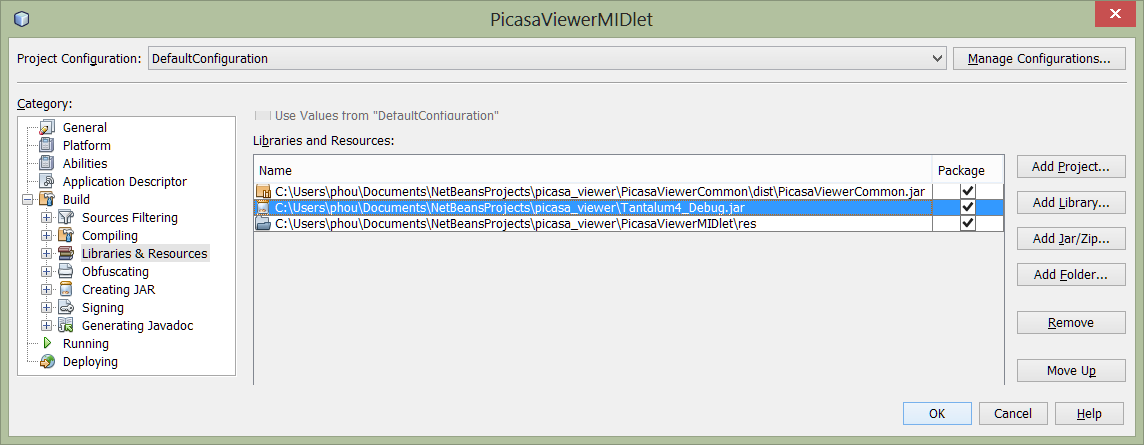
private final RSSModel rssModel = new RSSModel(40);  
 private final StaticWebCache feedCache = new StaticWebCache('5', new CacheView() {  
 public Object convertToUseForm(final LOR byteArrayReference) {  
 try {  
 final byte[] bytes = (byte[]) byteArrayReference.get();  
 rssModel.removeAllElements();  
 synchronized (Worker.LARGE\_MEMORY\_MUTEX) {  
 rssModel.setXML(bytes);  
 }  
 return rssModel;  
 } catch (Exception e) {  
 //#debug  
 L.i("Error parsing XML", rssModel.toString());

return null;  
 }  
 }  
 });

# Setting Up Your First Project

Let us start with the J2ME example, and continue with Android in a following section. For this and future examples we will illustrate with the Netbeans IDE, available at <http://netbeans.org/downloads/> . You may be more familiar with the Eclipse IDE, in which case you may find the pre-integrated version of Eclipse in the Nokia SDK 2.0 and later useful. See <http://www.developer.nokia.com/Develop/Series_40/>

Once you have brought in any existing source code you will use, **“Add JAR/ZIP…”** to bring the Tantalum library to your class path in **Project – Properties**.



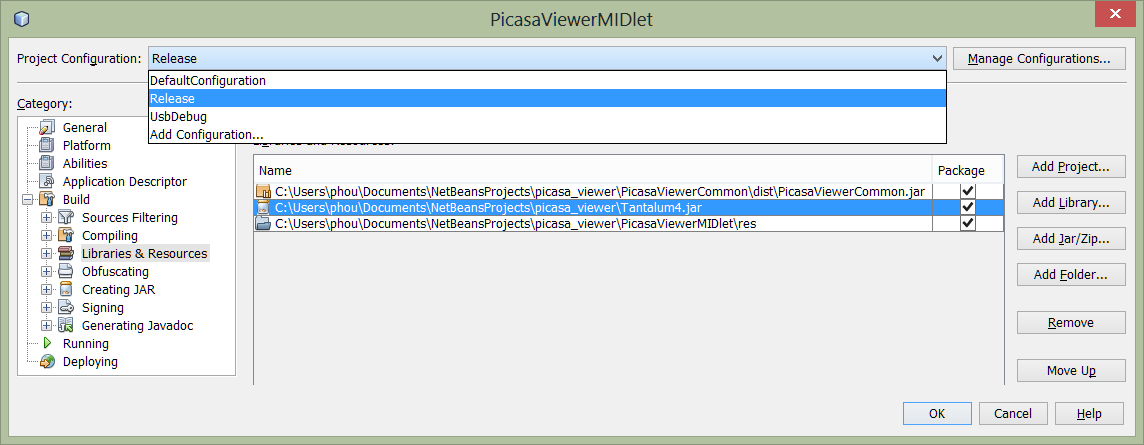
Note that there are three pre-build versions of the Tantalum library available:

1. **TantalumJME-debug.jar** – Most useful for development and debugging as it will list detailed information and error codes in your IDE’s debugger window when you are running in the emulator
2. **TantalumJME.jar** – This is the smallest package and should be used for your release build. No debugging information is generated, and the Proguard obfuscator has been applied to minimize and speed up the library code

When working the Android, use the associated Android JAR files instead. If you have installed the Tantalum library source code as a project in your IDE, you should refer directly to that project by using the **“Add Project…”** dialog instead. This is convenient as it allows you to step into the library and visit the source code implementation more easily.

## Creating Multiple Build Configurations

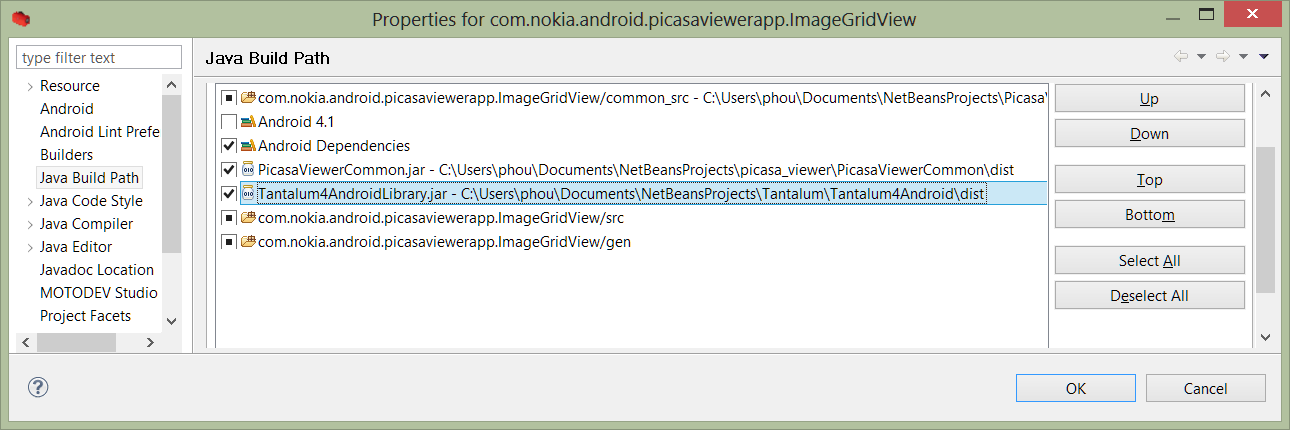
You may want to create multiple configurations in your IDE to easily switch between debug, on-phone USB debug, and high performance release builds. This is also useful for localization while keeping each package small. You do this by clicking “Add Configuration…” as shown below and then editing the libraries and possibly other resource files such as localized text and graphics included in the JAR file built by each configuration.



## Setting Up the Android Project

Once your project is running well in JME, you can refer to it from Android and directly re-use the business logic. Android will require you to re-create the user interface using Android classes, but this is not difficult.

You will need to refer to the Android packaging of the Tantalum library, and this is included in both source and pre-build JAR forms in your Tantalum4 distribution. Your Android project, usually created in Eclipse, should also refer to your application’s business logic as shown below (“PicasaViewerCommon.jar”). We will describe this in the next section.



## Separation of Application Logic and User Interface

If you plan to build cross-platform applications, a good start is to separate the cross-platform application logic into a different project from the JME user interface project. In the example above, the cross-platform application logic using the Tantalum library is only 2 classes in a project **PicasaViewerCommon.jar**. This Java 2 (JME) library is then referenced from Android and used unchanged when linked against the **TantalumAndroidLibrary.jar**. The two classes in the application logic are for contacting and parsing the Picasa web service JSON data feeds, and for representing those parsed results as a Java object. The remainder of the application is in two other projects: one for the J2ME UI, and another for the Android UI.

## A Word on Obfuscation

When you are minimizing the release build of your JME project, you may need to include the following Proguard parameters to help the pre-processor work past some of the cross-platform code even though it does not have the Android library as a dependency.

-dontwarn  
 -dontnotify

If you continue to face problems, it is easier to debug and tune the obfuscation when you include Android.jar in your build path.

# Summary

A lot of work has gone into keeping Tantalum small and making the powerful abilities of a modern mobile phone easy to access for creating a very fast and fluid user experience. This manual is a starting point to explore those capabilities, but the ultimate guide is the JavaDoc documentation, the complete example applications, and ultimately the constantly improving source code.

Feedback on this manual, requests for new features, and fresh hands willing to work are always welcome. Please contact us through either the project website or send an email to [paul.houghton@futurice.com](mailto:paul.houghton@futurice.com) . We hope this makes your mobile java development more fun and higher quality. Join us, and keep your code fast and tight.

# About the Author

Paul Houghton is a Nokia Certified Trainer, Nokia Champion and the Director of Wizardry and Development at Futurice. Futurice is a mobile and web application development consultancy with offices in Helsinki, Tampere, Berlin and London.

Paul has an MS in Industrial Engineering and Management and ExecutiveMBA from Helsinki University of Technology. He also has a BS in Electrical Engineering from Tennessee Technological University. Paul has founded or co-founded five companies in Finland, Great Britain and the United States before joining Futurice to help customers with coaching, training and solving really hard mobile problems. Paul travels frequently for training and speaking assignments. He currently lives in Berlin with his wife and two daughters. You can reach Paul at [paul.houghton@futurice.com](mailto:paul.houghton@futurice.com)