TANTALUM QUICK START MANUAL

All about getting started with the smallest, smartest mobile Java library for cross platform J2ME and Android application logic with advanced caching and simplified worker threading for fast, concurrent execution

Paul Houghton

Contents

Introduction

In November of 2010 Paul Houghton was teaching a motivated group of developers in Pretoria, South Africa and droning on about how mobile Java applications should be created and structured to create a great user experience (UX). The understandable response was “OK, show us”, and the challenge was laid. 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. This first release was the crystallization of over 10 years of concurrent and mobile Java application development experience has since been refined and developed by many hands into the current Tantalum 4 release discussed here.

It has been used very successfully in a number of commercial applications delivered by Futurice and other consulting companies. It makes your life easier, and release 4 expands this to help both beginning and advanced mobile developers that want to “do cross-platform right” on J2ME and Android. The approach is to strictly separate clean, concurrent, event-driven background application logic from clean, single thread event-driven UI code. This way you can achieve the best-possible user experience (UX) on each platform you support by using the best native UI classes while re-using your application logic across all platforms.

“Less is more” would be the best summary of our guiding philosophy. If fewer lines of code will do the job, that is usually cleaner to maintain and faster to execute. We love lambda expressions and do our best bring this type of modern method-pointer-passing front end architecture with advanced fork-join support to current mobile Java devices. We don’t try to fulfill every conceivable use case, just the most common ones that come up over and over again.

There is no user interface dependency in Tantalum- you can and should choose the best UI available for your purpose on each platform you target. You can freely extend the library to your specific purpose; that is why it is free and open. You no longer have to do the same boilerplate code for each project, nor do you need to re-write and extensively re-test the code for each platform that you port your code to.

The Tantalum Project Team

Tantalum is the combined work of many people led for better or worse by Paul Houghton and his long experience making really, really fast mobile applications that are a pleasure to use under varying network conditions. Significant contributions by Timo Saarinen, Oskar Ehnström, Samuli Saarinen, Vera Andersson, Mark Voit and others have been invaluable.

We would like to give a special, deep thanks to Nokia and Futurice for their generous and mostly willing sponsorship of this open source project. Without your patience, it would never have come into being. No one specifically asked for this library to be created or allocated a budget for it. Instead it results from the steady accumulation of ideas and spurts of free-time creativity along with various training projects and experiences and condensed here into a compact, reusable cross-platform library.

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 J2ME

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`.

Support for additional platforms just as J2SE Java7 with UIs such as JavaFX may be added in the future depending on interest. If you want to attempt this, simply copy the Android package and replace the few Android dependencies such as the SQLite database with Java7 equivalents.

License

Tantalum is licensed under the Apache 2 open source license, http://www.apache.org/licenses/LICENSE-2.0.html . This means you can freely include the code, as either a library or modified or unmodified source code, in your project. You cannot claim to be the inventor of this code except to the degree you have contributed to the project.

We kindly request you to give a word of thanks to the Tantalum project and provide in link in your About page along with any other similar licensed software which you have used in your product.

Getting the Source Code

You can find everything at Tantalum Project, http://projects.developer.nokia.com/Tantalum . This includes a BBC RSS Reader example application implemented using three different user interfaces: J2ME Forms, LWUIT, and J2ME Canvas using Nokia's proprietary Series40 extensions.

If you are developing for both J2ME and Android, a good cross platform example 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 Mercurial on your computer and then open a command prompt in your Netbeans project directory and type:

```
hg clone https://projects.developer.nokia.com/hg/Tantalum
```

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 Tantalum

In J2ME, the simplest initialization is to let your MIDlet extend the TantalumMIDlet class. You then need to call the super class constructor and tell Tantalum how many Worker threads you would like to use. Generally four is a good balanced number for most networked and cached applications.

```
public class RSSReader extends TantalumMIDlet implements CommandListener {
    /**
     * The RSSReader constructor.
     */
    public RSSReader() {
        super(4);
    }
```

You can close your program cleanly including the completion of any queued shutdown tasks by calling either `TantalumMIDlet.this.exitMIDlet(boolean unconditional)` or `Worker.shutdown(boolean unconditional)`. The two calls are equivalent. 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.

For Android applications, similarly have your main `Activity` extend `TantalumActivity`. And it will create four Workers automatically.

```
public class ImageGridView extends TantalumActivity implements
OnItemClickListener {
```

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 Worker.fork(Workable workable, int priority) where priority is one of the following:

1. `Worker.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(Workable workable)` without providing a priority.
2. `Worker.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. `Worker.LOW_PRIORITY` – The task will be added to a different queue, and only executed when there is nothing else to do in the main queue. No more than one LOW_PRIORITY task will run at a time.

This is useful for pre-fetching data from the server without contending for resources used by current user interface operations.

Most operations are from a single common task queue, however `Worker.forkSerial()` can be used to guarantee sequential execution by one of the Worker threads. `Worker.forkShutdownTask()` can be used to add work such as closing resources that will be completed before the program exits.

The simplest object which can be queued for background execution is a Workable. This does one thing, is stateless, and generally no feedback is given when the operation completes.

```
public interface Workable {
    /**
     * Do a task on a background thread, possibly returning a result for
     * asynchronous pipeline operations.
     * @param in
     */
    public Object exec(Object in);
}
```

The usage pattern is

```
Worker.fork(new Workable() {
      public Object exec(Object in) {
            // Do something on a background Worker thread
      }
});
```

`Workable` is a bit limited, so the more capable `Task` extends this. `Task` is often extended by your code to complete operations and return asynchronous results. An alternate usage pattern discussed in the fork-join section below is instead of extending the `Task.join()` the `Task` returned from an asynchronous method and then use the completed result.

Each `Task` has an internal state which starts as `Task.READY`. This changes automatically as the task is queued and executed. The states are:

- `Task.EXEC_PENDING` – The `Task` is queued but has not yet been pulled by a `Worker` thread.
- `Task.EXEC_STARTED` – A `Worker` has pulled the `Task` from the queue and is currently running the `exec()` method.
- `Task.EXEC_FINISHED` – The `Object exec(Object in)` method inherited from `Workable` complete normally on a background `Worker` thread
- `Task.UI_RUN_FINISHED` – The Task is a `UITask` which has also completed running on the UI thread
- `Task.CANCELED` – `cancel()` was explicitly called on the `Task` before the execution completed and the Task's `onCancelled()` method will be called on the UI thread

- `Task.EXCEPTION` – A runtime exception which could not be corrected occurred. `onCancelled()` will be called on the UI thread
- `Task.READY` – The initial state of all Tasks. A READY Task can be forked.

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's `final exec()` method will handle the state changes, and instead you place your code in an extension of the `doInBackground()` method.

```
Worker.fork(new Task() {
      public Object doInBackground(Object in) {
            // Do something on a background Worker thread
      }
}
```

An alternate syntax usage pattern is

```
(new Task() {
      public Object doInBackground(Object in) {
            // Do something on a background Worker thread
      }
}).fork();
```

Task also keep the result of the work for you, and this can be convenient if you want to use that result in a second method run on the UI thread. That is what the `UITask` extension of `Task` is for

```
 (new UITask() {
      public Object doInBackground(Object in) {
            // Do something on a background Worker thread
            return Image.createImage(in); // Task stores this "result"
      }
      public void onPostExecute(Object result) {
            // Do something on the UI thread using the Image "result"
      }
}).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. This feature is still experimental, and for now it is best to extend `Task` rather than work with chaining. Your thoughts and design input on this developing feature are useful- if you are interested, welcome to the Tantalum Project.

Networking

All this threading and `Worker` and `Task` is interesting, but what can we use it for? `HttpGetter` and `HttpPutter` are `Task`s that support the network HTTP_GET and HTTP_PUT 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 PUT 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.

```
final int HTTP_GET_RETRIES = 3;
HttpGetter httpGetter = new HttpGetter(url, HTTP_GET_RETRIES) {
      public Object doInBackground(final Object in) {
            // Perform an operation on byte[] "in" 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
      };
}
Worker.fork(httpGetter); // Queue for worker thread background execution
```

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$thumbnail").getJSONObjec
t(0).getString("url");
                final String imageUrl =
m.getJSONObject("media$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 Pattern Adapted for Rich UI Threading

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() {
            public Object doInBackground(final Object in) {
                  if (in != null && selectedImage ==
                              PicasaStorage.getSelectedImage()) {
                        image = (Image) in;
                        stopSpinner(); // Hide the image load spinner
                  }
                  return in;
            }
      }, Worker.HIGH_PRIORITY).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 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 DataTypeHandler associated with it, and this will automatically convert from the HTTP_GET result byte[] format into use form. In the example below, the DataTypeHandler parses the XML data in an RSS feed.

```
    private final RSSModel rssModel = new RSSModel(40);
    private final StaticWebCache feedCache = new StaticWebCache('5', new
DataTypeHandler() {
        public Object convertToUseForm(final byte[] bytes) {
```

```
            try {
                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. **Tantalum4_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. **Tantalum4_UsbDebug.jar** – Use this version of the library when running on a Nokia phone connected to your computer by USB cable. You can open a terminal emulator program on the computer and list

detailed information including performance profiling of concurrent tasks at full device speed which is often faster than emulator speed

3. **Tantalum4.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

If you have installed the Tantalum4 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.

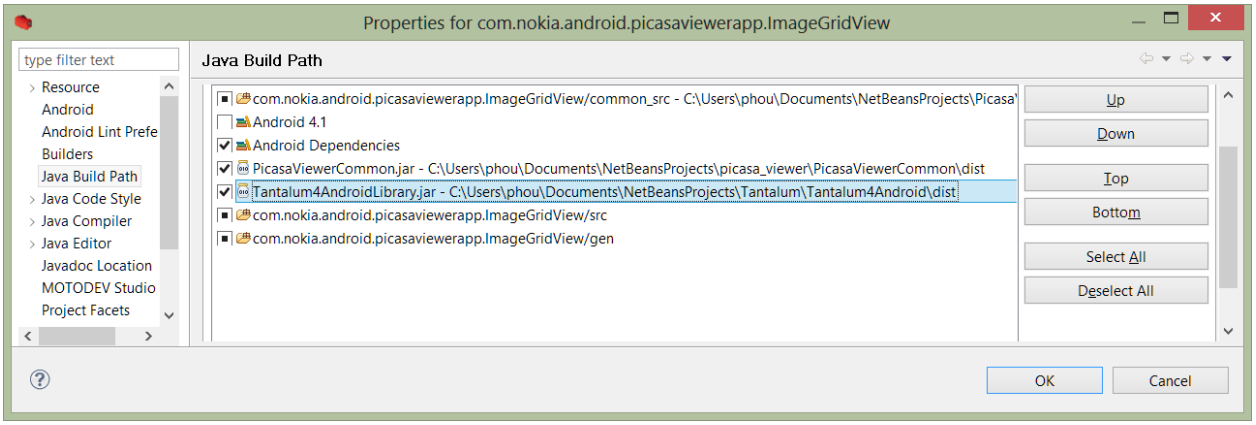
Creating Multiple Build Configurations

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 J2ME, 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 J2ME 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 (J2ME) 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.

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 . We hope this makes your mobile java development more fun and higher quality. Join us, and keep your code fast and tight.