Code Commenting

Guidelines on Writing Good Code Comments to be parsed by Doxygen

This page features tips and suggestions on writing good comments in header files to be parsed by Doxygen. We're running the initiative of getting a nice reference for DRLVM Interface Reference and Java class library reference, and our current experience shows that some specific conventions should be followed to get easily readable, complete and useful documentation. You are welcome to use, add or edit guidelines published on this page. We're not trying to impose a way of writing or a specific coding style, but rather agree on some common rules that would work for everyone.

Please note that the reference for DRLVM is already available on the website, and class library docs are (hopefully) soon to come. The guidelines below are for those who wish to improve the comments in source code and see how Doxygen parses them.

Here we go.

1. Prerequisites

We don't have a unified Doxygen reference building process at the moment. For example, you don't need Apache Ant to build the docs for classlib as you do for drlvm.

To be able to generate comments by using Doxygen, you will need to have the following:

- Checked out source tree with the files you want to parse.
- Doxygen version 1.5.* http://www.stack.nl/~dimitri/doxygen/
  The currently published doc bundles use the version 1.5.1. The problem lacks investigation, but it seems that 1.4.* and earlier versions do not support some valuable features.
- Apache Ant
- Doxyfile config file and build scripts: for DRLVM, located at `drlvm/trunk/vm/doc`.

With these, you should be able to run <code>ant</code> in vm/doc directory and build documentation successfully.

2. How To Write Comments

2.1 Ideology

Here I'd like to say a few words about where and how to create / group content in your headers. The major rule of distributing content in a header is:

RULES OF THUMB

- File definition at top of page, after the copyright and revision
- Element-specific definitions go immediately before the defined element; every element
- Generic info not belonging to any one element at the bottom of page

Now, these rules explained.

- **File definition at top of page, after the copyright and revision**

Please don't forget to add the `@file` tag to the head of your header. Give a run-in definition to the file content and a 2-3 line extended definition. With this, Doxygen includes the run-in definition to the list of files, and the extended definition to the file-specific page of documentation.

Example:

```
/**
 * @file Class Loader interface
 * Class loading functionality of the class support interface.
 * These functions are responsible for loading classes
 * from the virtual machine and interested components.
 */
```

if you don't get the desired detailed vs. brief definition, separate the run-in and the long definitions with a blank line.

- **Element-specific definitions go immediately before the defined element**

Leave element-specific content in the comment before the code element. Define each element: a function, a structure, a define, a typedef, an enumeration and each element inside. Many of these definitions might seem obvious, but what is obvious to you, might not be so clear to your readers across the world. And consistency always matters 🙂

- **Unite related code entities into groups.**

For example, you can group several functions inside an interface to show major areas that the interface covers. Definitions of grouped code entities will appear in listings of their respective headers. But try clicking a grouped element - WHOOSH! - you are on another page that lists all the code entities of this type. Doxygen creates such pages automatically, uses the group title as the page title and the group description as the page description. To create a group, use the `@defgroup` tag (immediately following the tag goes the unique group name, then the group title; on the next line goes the group description). To add a code entity to a group, add `@ingroup` group-name to the comment block before the code entity.

- **Place generic info not belonging to any one element at the bottom of page**

Store all generic content in one place: at the bottom of header. If you have more than two short paragraphs to say, better create one or multiple sections to organize your content (use the `@section` tag). You can also create headings for specific bits of info.

Entering a `@page` tag sets off this content to appear on a separate page. This functionality is somewhat analogous to JavaDoc's package.html file mechanism. However, with Doxygen, you do not need special files.

⚠ Be careful not to use the `@mainpage` tags more than once. This tag defines what appears on the starting page on the documentation, and Doxygen can only read and display `@mainpage` content from one file. We're now trying to write starting pages for all documentation bundles, and everybody's assistance is welcome.

Recommended samples

Class.h - nicely commented

Classes:

```
/** The constant pool of a class and related operations.
 * The structure covers all operations that may be required to run
 * on the constant pool, such as parsing and processing queries.*/

struct ConstantPool {
private:
    // tag mask; 4 bits are sufficient for tag
    static const unsigned char TAG_MASK = 0x0F;
    // this entry contains resolution error information
    static const unsigned char ERROR_MASK = 0x40;
...
    // constant pool size
    uint16 m_size;
    // constant pool entries; 0-th entry contains array of constant pool tags
    // for all entries
    ConstPoolEntry* m_entries;
...
public:
    /** Initializes the constant pool to its initial values.*/
    ConstantPool() {
        init();
    }
...
```

Functions

```
/** Checks whether the constant-pool entry represents the string of
  * the #CONSTANT_Utf8 type.
  * @param[in] index - an index in the constant pool
  * @return <code>TRUE</code> if the given entry is the <code>utf8</code>
  *         string; otherwise <code>FALSE</code>.*/
bool is_utf8(uint16 index) const {
     return get_tag(index) == CONSTANT_Utf8;
    }
```

Enumerations

```
/** Types of constant pool entries. These entry types are defined by a seperate
* byte array that the first constant pool entry points at.*/
enum ConstPoolTags {
    /** pointer to the tags array.*/
    CONSTANT_Tags               = 0,
...
```

compmgr.h - classes, handles

```
/**
 * @ingroup Handles
 * The handle of the abstract component interface.
 */
typedef const struct _OpenInterface* OpenInterfaceHandle;
```

hythread_ext.h

File definition, overview

```

/**
 * @file
 * @brief Extended Treading and synchronization support
 * @details
 * Thread Manager native interface. Provides basic capablitites for managing native threads in the system.
 *
 * <p>
 * <h2>Overview</h2>
 * The Thread Manager (TM) component provides various threading functionality for VM and class libraries code.
 * provides support for threading inside the virtual machine and for class libraries.
 * The implementation of thread management uses the black-box approach exposing two interfaces: Java and
native.
 * The whole component is divided into two layers: the middle layer of native threading with the layer of
 * Java threading on top. The implementation of Thread manager is based solely on the Apache Portable Runtime
(APR) layer.
 * <p>
 * The top layer provides the following functionality (see jthread, thread_ti.h):
 * <ul>
 * <li> Maps Java threads onto OS native threads
 * <li> Support kernel classes support
 * <li> Support JVMTI
 * </ul>
...
 */
```

Jitrino Runtime %3E General Execution Support - module page for a group of functions in an interface

Native Frames - related page

2.2 Formatting your Comments

Most useful things are already written: Sun's BKM's on writing comments. So far, for DRL sources we've followed these whenever we can. However, these BKMs are targeted at another tool: Javadoc, so not all of them apply. This section gives some additional tips and guidelines different from those suggested by Sun.

Tag specifics:

`@param`

- insert a dash ⛔ to separate the parameter name and definition
- indicate the parameter type `[in]/[out]`
- start the parameter definition with a small letter, no period (.) at the definition end

Example:
`@param[out] return_value - the pointer to the pointer to the return value`

`@return`

- start with a capital letter, include a period at the definition end
- for Boolean values, write in ALL CAPS and format with `<code></code>` tags

Example:
`@return <code>TRUE</code> if the class represents an enum. For Java 1.4, always returns <code>FALSE</code>.`

`@note, @see, @sa`

- use sparingly
- avoid using more than one of each in a comment; try rewriting the comment so that only the most prominent info stands out

More formatting:

- *italics* (`<i></i>`) for names of parameters inside a comment
- `monospace` (`<code></code>`) for all other code entities; optional for cross-references
- cross references to related code entities; if the link is not placed automatically, use # to force a link

... This is what we have for now. Everyone is welcome to share experience or question any of the rules currently present.