YARD Release 0.1a (Feb. 24 2007)

SYNOPSIS

YARD is a documentation generation tool for the Ruby programming language (http://www.ruby-lang.org). It enables the user to generate consistent, usable documentation that can be exported to a number of formats very easily, and also supports extending for custom Ruby constructs such as custom class level definitions. Below is a summary of some of *YARD*'s notable features.

FEATURE LIST

1. RDoc/SimpleMarkup Formatting Compatibility

YARD is made to be compatible with RDoc formatting. In fact, *YARD* does no processing on RDoc documentation strings, and leaves this up to the output generation tool to decide how to render the documentation.

2. Yardoc Meta-tag Formatting

Like Python, Java, Objective-C and other languages, *YARD* uses a `'@tag'` style definition syntax for meta tags alongside regular code documentation. These tags should be able to happily sit side by side RDoc formatted documentation, but provide a much more consistent and usable way to describe important information about objects, such as what parameters they take and what types they are expected to be, what type a method should return, what exceptions it can raise, if it is deprecated, etc.. It also allows information to be better (and more consistently) organized during the output generation phase. Some of the main tags are listed below:

Table 1. Meta-tags and their descriptions

Tag Name and Arguments	Description
`@param [Types] name description`	Allows for the definition of a method parameter with optional type information.

Tag Name and Arguments	Description
`@yieldparam [Types] name description`	Allows for the definition of a method parameter to a yield block with optional type information.
`@yield description`	Allows the developer to document the purpose of a yield block in a method.
`@return [Types] description`	Describes what the method returns with optional type information.
`@deprecated description`	Informs the developer that a method is deprecated and should no longer be used. The description offers the developer an alternative solution or method for the problem.
`@raise class description`	Tells the developer that the method may raise an exception and of what type.
`@see name`	References another object, URL, or other for extra information.
`@since number`	Lists the version number in which the object first appeared.
`@version number`	Lists the current version of the documentation for the object.
`@author name`	The authors responsible for the module

You might have noticed the optional "types" declarations for certain tags. This allows the developer to document type signatures for ruby methods and parameters in a non intrusive but helpful and consistent manner. Instead of describing this data in the body of the description, a developer may formally declare the parameter or return type(s) in a single line. Consider the following *Yardoc*'d method:

```
##
# Reverses the contents of a String or IO object.
#
# @param [String, #read] contents the contents to reverse
# @return [String] the contents reversed lexically
def reverse(contents)
  contents = contents.read if respond_to? :read
  contents.reverse
end
```

With the above `@param` tag, we learn that the contents parameter can either be a String or any object that responds to the 'read' method, which is more powerful than the textual description, which says it should be an IO object. This also informs the developer that they should expect to receive a String object returned by the method, and although this may be obvious for a 'reverse' method, it becomes very useful when the method name may not be as descriptive.

3. Custom Constructs and Extending YARD

Take for instance the example:

```
class A
  class << self
    def define_name(name, value)
      class_eval "def #{name}; #{value.inspect} end"
    end
  end

  # Documentation string for this name
  define_name :publisher, "O'Reilly"
end
```

This custom declaration provides dynamically generated code that is hard for a documentation tool to properly document without help from the developer. To ease the pains of manually documenting the procedure, *YARD* can be extended by the developer to handled the '`define_name`' construct and add the required method to the defined methods of the class with its documentation. This makes documenting external API's, especially dynamic ones, a lot more consistent for consumption by the users.

4. Raw Data Output

YARD also outputs documented objects as raw data (the dumped Namespace) which can be reloaded to do generation at a later date, or even auditing on code. This means that any developer can use the raw data to perform output generation for any custom format, such as YAML, for instance. While *YARD* plans to support XHTML style documentation output as well as command line (text based) and possibly XML, this may still be useful for those who would like to reap the benefits of *YARD*'s processing in other forms, such as throwing all the documentation into a database.

Another useful way of exploiting this raw data format would be to write tools that can auto generate test cases, for example, or show possible unhandled exceptions in code.

USAGE

Currently *YARD* only is usable to the client via a quick and dirty implementation of '`ri`' called `yri` (packaged as '`yri`'). Execute '`yri`' in the root directory of your codebase to have *YARD* generate documentation for your code. '`yri`' is only meant to work for method definitions, though it returns (irrelevant) information for classes and other objects too. The syntax is:

```
./yri Module::With::Class#and_method_name
```

For example, you can use '`yri`' to show object from *YARD*'s codebase by executing in the yard root directory:

```
./yri YARD::CodeObject#attach_source
./yri RubyLex::BufferedReader#ungetc
```

The first command shows how tags are added to the object and how blocks can be introspected, while the second command shows how *YARD* can handle undocumented exceptions and document them anyway.

NOTES FOR RELEASE 0.1a

This release is highly experimental and should only be used for testing purposes only. It is likely to break with unconventional code styles or large projects. Testing has mainly been from the *YARD* codebase itself and a small other project that had been written with *Yardoc* formatted documentation, but I expect a lot of problems with other code. Please inform me of your results

CHANGELOG

- Feb.24.07: Released 0.1a experimental version for testing. The goal here is to get people testing YARD on their code because there are too many possible code styles to fit into a sane amount of test cases. It also demonstrates the power of YARD and what to expect from the syntax (Yardoc style meta tags).

COPYRIGHT