TCode

User Guide

Tools for documenting Java programs in LaTeX

Version: September 17, 2008

This document describes facilities that help writing Java classes, together with detailed documentation of their application programming interface (API), in a single LaTeX file. The LaTeX package `tcode` offers special commands and environments for typesetting API documentation and user guides. It permits one to display certain parts of the code (e.g., method headers), hide others (e.g., method bodies and private variables), and explain the methods, variables, etc., in a uniform format. Although it is targeted to that language, it works not only for Java, but also for other programming languages such as C, C++, etc. LaTeX will produce the documentation as a `.dvi`, `.ps`, or `.pdf` file.

A Perl script called *texjava* produces a `.java` file by extracting the code form the `.tex` file. With the help of LaTeX2`HTML`, *texjava* also transforms some of the documentation into Javadoc format, so that the *javadoc* program can later produce HTML documentation from the `.java` file. Another Perl script called *javatex* does the reverse transformation, recovering the `.tex` file from the `.java` file.

1 Introduction

Javadoc is the standard Java tool for generating API documentation [2]. Its output is in HTML format. However, HTML is very limited on certain aspects such as displaying mathematical expressions, complex tables, etc. Good modern typesetting systems such as TeX and LaTeX offer much better quality, flexibility, and precision than HTML for producing detailed and nice-looking documentation [3]. LaTeX is certainly a tool of choice for API documentation of mathematically-oriented software. On the other hand, for Java libraries, standard online HTML documentation based on Javadoc is almost a necessity.

With the tools described here, one can program a Java class and document its API in great detail, in a single LaTeX file. The LaTeX program produces a nicely-typeset version of the documentation, while a Perl script called *texjava*, helped by LaTeX`2HTML` [3], produces a `.java` file, which can be compiled into a `.class` file and can also be used by javadoc to produce HTML documentation. The `.java` file produced by *texjava* can be retransformed (to a certain extent) into a LaTeX file by the Perl script *javatex*.

The LaTeX package `tcode` defines special commands and environments for typesetting API documentation in a uniform format, and selectively displaying or hiding (from the API documentation or user's guide) pieces of Java code. This LaTeX class file can also be used for other programming languages such as C, C++, etc. In this document, we assume that the target language is Java.

These tools are rudimentary and much simpler, for example, than the CWEB system of documentation [4], also based on TeX.

Section 2 explains how LaTeX files should be organized for using these tools, describes the main commands, and give an example. Section 3 shows how to run *texjava*. Sections 4 and 5 tell about *javatex* and how to run it.

2 Layout of the LaTeX files

A LaTeX document using the `tcode` LaTeX package to describe a Java package (or set of packages) normally has a single master file, with several secondary `.tex` files, which are loaded via `\include{ }` commands in the master file. There should be one secondary `.tex` file for each `.java` file that we want to have (typically, this is one file for each Java class or interface in the package). The master file may contain other material and include additional files as well. LaTeX or pdfLaTeX will process the master file to produce the detailed documentation, which will usually end up as a `.pdf` file. The `.tex` files that implement Java classes are transformed one by one into the corresponding `.java` files (or `.c`, `.cpp`, etc., in the case of other languages) by the Perl script `texjava.pl` helped by the LaTeX`2HTML` program, which basically extracts the code and formats additional information for *javadoc*. Since the script can process only one `.tex` file at a time, one may create a Makefile to automate the generation process when dealing with many files.

The `tcode` package. To use the commands of the `tcode` LaTeX package, this one must be imported via the command

```
\usepackage{tcode}
```

This command will locate and load `tcode.sty`, so this file must be accessible in a directory pointed to by the `TEXINPUTS` environment variable. The `tcode` package requires the `html` and `alltt` packages which should also be locatable through the `TEXINPUTS` environment variable.

The code environment. The basic construct for identifying programming code in the LaTeX files is the `code` environment. Everything between `\begin{code}` and `\end{code}`, with the exception of LaTeX commands, is displayed in special verbatim mode, with a typewriter font, by LaTeX and is treated as Java code (copied into the `.java` file) by *texjava*. The Perl script disallows nesting of `code` environments. The `\end{code}` command should always be placed alone at the beginning of a line. A document is therefore composed of an alternance of text blocks and code blocks, starting and ending with a text block.

The contents of a `code` environment is actually treated as a "box" by LaTeX, which means that all the code it contains will appear on the same document page. This is what we expect when documenting methods, because we wish a multi-line signature not to span on two pages. However, if the code is too long to fit in the page, LaTeX will issue an "overfull vbox" warning and the code will not appear entirely on the resulting document. The `longcode` environment is convenient for displaying long blocks of code; it behaves just as the `code` environment, except that it allows the code to span over two or more pages.

There are situations where one would like to have some code displayed in the documentation, but not have it in the `.java` file (e.g., when giving programming examples in the documentation). The `vcode` and `longvcode` environments do that: LaTeX treats them just like the `code` and `longcode` environments, but *texjava* does not copy the code into the `.java` file. However, it will appear as typed text in the HTML documentation.

The following character strings should never appear inside programming code (e.g., in a litteral string or in a comment), because they will be interpreted as active LaTeX commands: `\hide`, `\endhide`, `\code`, `\endcode`, `\begin{hide}`, `\end{hide}`, `\begin{code}`, `\end{code}`. If one needs to use such a string, one must separate it in two parts, as in the following example: `String s = "\\" + "begincode";`.

The hide environment. Normally, we do not want all the code to be displayed in the API documentation. For example, the private fields, methods, and classes, as well as the method implementations, should be hidden. The `hide` environment permits one to easily hide parts of the code and text from the LaTeX documentation: Everything between `\begin{hide}` and `\end{hide}` is simply not shown in the document produced by LaTeX. This environment can be applied to a block of text as well as to a block of code (i.e., inside the code environment). Nesting is allowed, but not inside the code environment. For example, one may place a whole set of methods together with their descriptions in a `hide` environment, to remove them from the documentation, while part of the code of each method may already be in a `hide` environment (inside the `code` environment). In this case, the outer `hide` environment must begin and end *outside* any `code` environment.

The `hide` environment has no effect on the code extracted by *texjava* in the sense that all hidden code goes to the `.java` file anyway. However, the hidden documentation will not appear in the HTML conversion produced by Javadoc.

No hiding. In case one wishes a documentation that contains all the code, including the method implementations and other private material, one can simply turn OFF the hiding mechanism of the `hide` environment with the command `\hidefalse`. It can be turned ON again with the command `\hidetrue`. However, turning the hide mechanism OFF will get LaTeX to parse hidden contents to display it. It will then react to backslashes in the hidden code and try to interpret control sequences. For example, if some hidden code contains

```
System.out.println ("This is a string,\nAnd another one.");
```

it must remain hidden otherwise LaTeX will complain about the invalid control sequence `\n`.

Detailed contents. The `detailed` environment is convenient for producing two versions of the documentation, one more detailed than the other. It comes with a switch that can be turned ON by the command `\detailedtrue` and turned OFF by the command `\detailedfalse`. Everything inside the `detailed` environment is ignored by LaTeX when the switch is turned OFF, otherwise it is included normally. By default, the switch is turned ON. When creating the HTML version, the `detailed` environment is ignored, meaning that the HTML document will never be detailed.

Restrictions due to LaTeX2HTML. The *texjava* script calls the LaTeX2`HTML` utility to convert the documentation in the `.tex` file to HTML format in the `.java` file. It generates HTML 4.0 code using Unicode for mathematical symbols. This utility processes standard LaTeX commands, but not user-defined commands/macros and commands/macros found in external packages. Adding support for new packages implies writing Perl code intended for processing the different commands provided by these packages [3]. This operation is complex and requires a good understanding of the LaTeX2`HTML` conversion process (which is yet undocumented at the time of this writing).

The contents intended for HTML conversion should therefore be as simple as possible and should contain only standard LaTeX commands and environments. Unknown environments and mathematics are translated to images using LaTeX (called by LaTeX2`HTML`). However, image creation is time consuming and should be avoided by restricting usage of the math mode and complex environments to *LaTeX-only* subblocks (see below). Sometimes, images may mysteriously appear with a gray background. If LaTeX generates an error when making an image, it will skip it, LaTeX2`HTML` will not notice that, and subsequent images may have the wrong number. The `equation` and `eqnarray` environments are converted to their star forms because problems arise with equation numbers. The `ref` and `cite` commands are also ignored because reference collecting accross several independent runs of LaTeX2`HTML` causes problems.

Except for the `hide` environment, no other environment should span more than one documentation block, i.e., should not contain a `code` environment. Departure from that rule could prevent LaTeX2`HTML` from writing the markers separating the blocks (i.e., the fields and methods), and empty documentation blocks could then appear for subsequent fields and methods. The script calls LaTeX2`HTML` only once for the LaTeX file given to it, so any modification of the LaTeX parameters in one text block could have an impact on the subsequent blocks.

LaTeX-only and HTML-only parts. Certain parts of the `.tex` files can be intended only for LaTeX, others only for the HTML file. This can be specified by the usual commands and environments available in the `html` package (e.g., the `latexonly`, `htmlonly`, and `rawhtml` environments).

Everything inside a `latexonly` environment is ignored by LaTeX`2HTML`. This is a good place for complex mathematical expressions and tables, for example. On the other hand, everything inside a `htmlonly` or `rawhtml` environment is ignored by LaTeX. Material inside a `rawhtml` environment is assumed to be HTML code, which is copied directly to the `.java` file and eventually the `.html` file, whereas material inside the `htmlonly` environment is translated into HTML by LaTeX`2HTML`.

Header of a class or module. The command `\defclass{`*classname*`}` can be used to start the documentation of a Java class or interface. The command places its argument on a line by itself, centered, in a large font. It also adds a corresponding entry to the table of contents and modifies the page headings to contain the class name. The command `\defmodule{`*name*`}` has exactly the same effect.

The first block of text is considered as a class or file documentation block. In the `.java` file, it will be inserted as a doc comment for *javadoc*, at the beginning of the class definition. For this reason, the first block of code processed by *texjava* must contain the Java class definition.

Subsequent blocks of code and text. After the first block of code (i.e., `code` or `longcode` environment), there is an alternance of a block of text, a block of code, a block of text, etc. For each such block of code, the text that follows it immediately is assumed to be its corresponding LaTeX documentation. Therefore, *texjava* will insert this text as a (javadoc) *doc comment* for the last field or method appearing in this block of code (and not in a `hide` environment).

When a class has many documented methods, it may be convenient to partition them into different groups in the LaTeX documentation, with a descriptive header above each group. The command `\guisec{`*header*`}` outputs such a header, centered horizontally, with horizontal lines on each side. It should not be placed inside a `code` environment and should always be alone in its own paragraph. A text block that contains this command will be splitted in two parts. The contents preceeding the section will be inserted as usual whereas the sectioning command and all the following contents will be inserted *after* the previous code block. With the `\unmoved` command, one can achieve the same effect of a sectioning command without displaying anything. Note that *javadoc* will always ignore this `\guisec` and any other sectioning commands, so it cannot be used to regroup class members in the HTML documentation.

Indenting documentation. The environments `tab`, `tabb`, and `tabbb` have been defined to indent the documentation and put it in a smaller font. These environments are normally used to describe fields and methods, they are ignored by LaTeX`2HTML`, and produce the following results:

> This is a text indented with `tab`.
>
> > This text is more indented because it is in a `tabb` environment.

This text is still more indented because it is in a `tabbb` environment.

Doc-comment tags for javadoc. The following LaTeX commands are mapped by *texjava* to the corresponding *javadoc* doc-comment tags which are used to encode specific information about classes, fields, methods, etc. (see [2, chapter 7]). These doc comments will also appear in the LaTeX documentation. For doc comments that are to appear only in the java and HTML files, one can use the *javadoc* tags directly (provided that *texjava* is called with the `-nosavelatex` option).

The LaTeX commands `\param`, `\return`, etc., are mapped to the doc-comment tags `@param`, `@return`, etc. These commands should be placed at the end of the method documentation, in the same order as they are described below. The available commands and their descriptions are:

`\param{`*param-name*`}{`*description*`}`

Gives a description for the parameter *param-name* of the current method. This parameter name must be a single word whereas the description is usually a short sentence. Such a description should appear for every method parameter.

`\return{`*description*`}`

Describes, in one or more sentences, what is returned by the current method. This should appear for every method returning values.

`\exception{`*class-name*`}{`*description*`}`

Gives the class name of an exception raised by Java when the situation described by the sentence in the second argument occurs.

`\class{`*class-name*`}`

Indicates the name of a class. This is formatted in typed text in the LaTeX document and converted to an hyperlink in the HTML document.

Example: `See class \class{GofStat} for more information`
`Use \class{java.util.List} to store the data.`

`\externalclass{`*package*`}{`*class-name*`}`

Indicates the fully qualified name of a class. This is formatted in typed text in the LaTeX document, and converted to an hyperlink in the HTML document. This macro can be useful if one does not want the fully qualified package name to appear in the LaTeX and HTML document whereas it is sometimes necessary for Javadoc to hyperlink to the class. If one wants to have the full package name, he can write the full qualified name as the argument of the `class` command.

Example: `The \externalclass{umontreal.iro.lecuyer.util}{Num}` will show `Num` as a label whereas `\externalclass{umontreal.iro.lecuyer}{util.Num}` will show `util.Num`.

`\method{`*method-name*`}{`*signature*`}`

Indicates the name of a method in the current class. This is formatted in typed text in the LaTeX document, and converted to an hyperlink in HTML. If the signature is empty, Javadoc will link to the first method with that name. The signature will never be shown in the link labels. If one wants to show the signatures, he must write it in parentheses after the method name. One can also use `\method` and the following `\externalmethod` to link to fields. In this case, the signature argument will be empty.

Examples: `\method{density}{}` and `\method{density}{double}` will generate the `density` label whereas `\method{density(double)}{}` will generate `density(double)`.

`\externalmethod{`*package*`}{`*class-name*`}{`*method-name*`}{`*signature*`}`

Indicates the name of a method in another class. This is formatted in typed text in the LaTeX document, and converted to an hyperlink in HTML. The package name, class name and the signature will not appear in the label name.

Example: `\externalmethod{umontreal.iro.lecuyer.gof}{GofStat}{andersonDarling}{}` will typeset `andersonDarling`

`\clsexternalmethod{`*package*`}{`*class-name*`}{`*method-name*`}{`*signature*`}`

This is the same as `\externalmethod` except that the class name is prepended. Since this macro can generate long strings in typed text which can easily result in overful boxes, a discretionary hyphen is added between class name and method name.

Example:
`\clsexternalmethod{umontreal.iro.lecuyer.gof}{GofStat}{andersonDarling}{}` will be formated `GofStat.andersonDarling`
`\clsexternalmethod{umontreal.iro.lecuyer}{gof.GofStat}{andersonDarling}{}` will format `gof.GofStat.andersonDarling`

An example.

```
\defclass{Complex}

This class allows one to work with complex numbers of
the form $a + bi$, where $a$ is the \emph{real} part of
the number, \emph{b} is the imaginary part and
$i=\sqrt{-1}$.

\bigskip\hrule\bigskip

\begin{code}

public class Complex\begin{hide} {
   private double realPart;
   private double imagPart;\end{hide}

   public Complex (double realPart, double imagPart)\begin{hide} {
      this.realPart = realPart;
      this.imagPart = imagPart;
   }\end{hide}
\end{code}
\begin{tabb}  Constructs a new {\tt Complex} number.
\param{realPart}{The real part corresponding to $a$}
\param{imagPart}{The imaginary part corresponding to $b$}
\end{tabb}
\begin{code}

   public Complex add (Complex c)\begin{hide} {
      realPart += c.realPart;
      imagPart += c.realPart;
      return this;
   }\end{hide}
\end{code}
\begin{tabb}  Adds two complex numbers.
  \param{c}{The complex number to add to this one.}
  \return{This object, allowing to perform more than one operation
    on a single line of code.}
\end{tabb}
\begin{code}

   // other methods...

   public String toString()\begin{hide} {
      return "(" + realPart + " + " + imagPart + "i)";
   }
}\end{hide}
\end{code}
\begin{tabb}   Converts the number to a {\tt String} of
  the form {\tt a + bi}.
\end{tabb}
```

3 Running Texjava

The Perl script `texjava.pl` converts LaTeX documents into Java code with javadoc-style comments. Everything inside a `code` or `longcode` environment is considered as Java code. Since LaTeX has a better error-detection scheme than *texjava* and LaTeX2HTML, it is a good idea to compile the documents first with LaTeX, before using *texjava*.

The LaTeX commands defined in the `tcode` package are implemented for LaTeX2HTML in the `tcode.perl` file. This file should therefore be accessible to LaTeX2HTML, i.e., it should be in a directory known by the `$LATEX2HTMLSTYLES` environment variable. See [3, page 99] for details about this. The script has been tested on Perl 5.8.0 but it should work with any version 5 Perl interpreter. Under non-Unix operating systems, such as Microsoft Windows, it is necessary to install a Perl distribution, such as ActivePerl [1], before using `texjava.pl`. For Linux environments using C-Shell, `Tcoderc` shell script can be used to set the environment. It needs the environment variable `TCODEHOME` to be set. It sets the `TEXINPUTS` variable, creates `.latex2html-init` if it does not already exist, and defines a `texjava` alias to allow running `texjava.pl` more easily.

The following command runs the script:

`perl texjava.pl` [`-(no)images`] [`-(no)html`] [`-(no)savelatex`] [`-htmloutdir` *dir*] [`-master` *masterfile*] [`-htmlonly`] [`-htmltitle` *title*] *infile* [*outfile*]

Here, `texjava.pl` has to be replaced by a path to the script if it is not executed in the `tcode` directory. Its arguments and options are as follows:

infile

The name of the input file, which should have the `.tex` extension. If the extension is not given and no file with such a name exists, `.tex` is appended to the file name. The input file is parsed but not modified.

outfile

The name of the output file. Normally, it should be a program file, e.g., with the `.java` extension. If no extension is specified, `.java` is assumed. When using the HTML option, `.html` is used. If no output file is given, the name of the input file is taken, with a `.java` or `.html` extension. The file will be created or replaced in the same directory as the input file. If *outfile* already exists, it will be replaced without notice, otherwise the file will be created. If *infile* and *outfile* are the same, an error will occur.

`-(no)images`

A switch to enable or disable image generation by LaTeX2HTML. With the `-images` option, images are generated and stored in the HTML output directory. They are copied into a subdirectory corresponding to the package of the Java file. With the `-noimages` option, the conversion can be much faster, but no images (e.g., complicated mathematical formulas) will appear in the HTML file and if it does not already exist, the HTML output directory will not be created. Images will be replaced by placeholders in the generated files. Default value: `-noimages`.

`-(no)html`

With the `-html` option, LaTeX2HTML is invoked to convert from LaTeX to HTML. With the `-nohtml` option, LaTeX2HTML is not called, so the script can run faster, but no HTML conversion of the LaTeX documentation is produced. This can be useful for environments which do not have LaTeX2HTML installed. The LaTeX contents is nevertheless saved as HTML doc comments in the Java code if `-savelatex` is passed. Default value: `-nohtml`.

`-(no)savelatex`

With the `-savelatex` option, the script will save LaTeX contents into Java doc comments, using special encoding. This results in rather ugly Java files, but permits one to recover the original LaTeX file (at least to some extent) from the `.java` file. The `-nosavelatex` option unclutters the output file, but prevents reverting to LaTeX. The combination `-nohtml -nosavelatex` disables insertion in the output file of all doc comments generated from the LaTeX source. Default value: `-nosavelatex`.

`-htmloutdir` *dir*

Specifies where the HTML output will be placed when *Javadoc* processes the Java files. This will indicate to `texjava.pl` where to put the image files generated by LaTeX2HTML. If not specified, the images will be copied into an `html` subdirectory of the output directory where the Java file is created. This option has no effet if the `-html` option is not given. The HTML output directory is automatically created unless it already exists or the `-noimages` option is specified. It is never emptied or removed by `texjava.pl`.

`-master` *masterfile*

The name of a master LaTeX file for the *infile* file. This file should compile successfully with the command `latex master`. Usually, it will contain the `\begin{document}` command and include the *infile* file. Its preamble may contain macros used in *infile*. The script will read only the preamble, i.e., everything before `\begin{document}`. This preamble is used when constructing the document intended for LaTeX2HTML processing. If no master file is specified, it is assumed that *infile* itself is the master file. Forgetting to specify the master file when it is needed could prevent some required packages from being loaded into LaTeX2HTML and generate unwanted additional images in the HTML file.

`-htmlonly`

Indicates that the script only has to convert a LaTeX document to HTML, not processing Java code blocks. This option is useful to convert overviews into `package.html` files. It is simpler and better than calling LaTeX2HTML directly because `texjava.pl` passes a bunch of options to LaTeX2HTML and transforms the `.tex` and `.html` contents to avoid some images from math formulas.

`-htmltitle` *title*

When using HTML-only mode, allows to give a title to the generated HTML file. This title will be placed in the `TITLE` element of `HEAD`. If one wants to have a title containing spaces, it must be surrounded by quotation marks for the shell to pass it as one argument to the script. If no title is given, an empty string will be used as the title for the HTML document.

Examples. To extract *Java* code from the LaTeX file `Event.tex`, assuming that the master file is `guide.tex`, and place the result in the file `Event.java`, one can use:

```
perl texjava.pl  -master guide.tex  Event.tex
```

which will create the `Event.java` file and, if needed, a `html` subdirectory that will contain images. To generate the *javadoc* documentation (in HTML), one can use a command of the form:

```
javadoc -d html  javafilesorpackages  Event.java  javafilesorpackages
```

As another example, to extract the *C* code from the LaTeX file `chrono.tex` and place it in the header file `chrono.h`, assuming that the preamble is in `guide.tex`, one can use:

```
perl texjava.pl  -master guide.tex  chrono.tex  chrono.h
```

Here, one must not forget to specify the target name otherwise `chrono.java` would be created instead of `chrono.h`.

Restrictions, limitations, and special cases.

- Due to the structure of the `code` environment, every LaTeX command taking no argument will be interpreted. Braces delimiting arguments will however display as contents in the code. This cannot be avoided because LaTeX must interpret `\begin` and `\end` to support contents hiding and `code` environment termination.

- If a package declaration is found before the class declaration in a given Java file, the script will place generated image files in an appropriate subdirectory of the HTML output directory. For exemple, if one writes the line `package a.b.c;` in its Java program, `texjava.pl` will copy the images into a subdirectory named `a/b/c`. This reflects the Javadoc generated structure and allows proper image retrieval by the browser.

- Althought the script allows white spaces within commands (e.g., `\begin {code}`), it can cause problems to LaTeX. Such spaces must be avoided in commands controling the `hide` environment inside a `code` environment, and in the `\end{code}` command.

- When hiding method bodies, one must add at least one space between `\begin{hide}` and the open brace. If one writes

  ```
  \begin{hide}{
  ```

  LaTeX will treat the rest of the file as an argument to the `hide` environment and an error will occur.

- The method bodies should always be hidden, otherwise, the insertion point of the documentation may be incorrectly computed.

- The first sentence of a doc comment is used by *javadoc* as a *brief* (a one-sentence summary) of the corresponding class, interface, field, or method. The first sentence of a LaTeX documentation block which is to be converted to a doc comment should therefore be short and simple. It should not contain special commands, mathematical formulas, etc., and should not be in a `latexonly` environment. If a formatting problem arises with a too complicated first sentence, the script will simply insert a period on a single line (to represent an empty brief). This artefact will appear in the *javadoc* documentation, but it will prevent badly formated HTML on the rest of the concerned page.

- With the `-savelatex` option, the strings `/*HIDE*/`, `/*ENDHIDE*/`, and `/**` will be converted to `/* HIDE */`, `/* ENDHIDE */`, and `/* *` everywhere. This implies in particular that *javadoc* comments already in the code will be ignored!

- If a `package`, `class` or `interface` keyword is found in a long comment before the Java class or interface declaration, and if there is no star preceeding the keyword on the line, the script will get confused about where to insert the class documentation or will put images at the wrong place.

- Problems may arise for Java because of the saved LaTeX contents. If a text block contains a LaTeX command starting with `u`, e.g., `\unitlength`, it will be replaced, in the comments, by `\@unitlength`. When reverting to LaTeX, this substitution will be undone by *javatex*, so one has to worry about it only when modifying the (hidden) saved LaTeX contents directly in the `.java` file before reverting to LaTeX.

- If no `code` or `longcode` environment is found in the input file, the output file will contain only a single doc comment.

- The script uses HTML 4.0 with Unicode for mathematical symbols. For the symbols to be displayed correctly, one should use a recent version of major browsers.

- To perform the conversion, a recent version of LaTeX2HTML is required. Version 2002 (1.62) does not convert greek letters to Unicode properly unless we use MathML which is not compatible with major browsers, whereas version 2002-2-1 (1.70) works properly.

4 The Javatex Script

Warning: This script is experimental and not very reliable, so one should use it with care. It is strongly recommended to make a backup copy of the original LaTeX file before using it. The script assumes that the code intended for conversion is syntactically correct and can be compiled without errors.

The Perl script `javatex.pl` can convert Java source files to LaTeX documents. The `.java` source files may have been produced by *texjava*, but not necessarily. For example, one may want to use a special editor for Java programs to edit the Java file and encode the LaTeX documentation in it by hand, in the same format as it would have been saved by *texjava*.

The Java source file intended for conversion should normally contain Java doc comments, of the form `/**` *comments* `*/`, where *comments* can be any (multi-line) text. The contents of these comments is converted into LaTeX documentation blocks for classes, interfaces, fields, and methods. Other types of comments in the code are simply ignored and preserved intact into code blocks.

The contents of HTML commented blocks of the form `<!--LATEX ... -->` will be inserted directly into the LaTeX document. This construct allows hiding LaTeX code inside Java doc comments. Note that a doc comment should never begin with such an HTML comment, otherwise *javadoc* could get mixed up and produce badly-formatted documentation. If a block contains only LaTeX code, it can be entered as a comment of the form `/*LATEX ... */` rather than as a doc comment, in order to avoid confusion of *javadoc*.

The `/*HIDE*/` ... `/*ENDHIDE*/` constructs found in the code are converted respectively to `\begin{hide}` ... `\end{hide}` constructs.

The `/*CODE*/`, `/*SMALLCODE*/`, and `/*LONGCODE*/` constructs are converted to the `code`, `smallcode`, and `longcode` environments.

All java doc-comment markers `/**` and `*/` will be removed, since these doc comments are converted to LaTeX documentation. Any star at the beginning of a line in a doc comment will also be removed to follow Java doc-comment formatting conventions. Doc comment are considered converted from *texjava* and should contain some special indications. Since HTML to LaTeX conversion is not supported, unmarked text in comments is simply discarded.

The first doc comment is considered as the class or interface documentation and will appear at the top of the resulting document. In the code, it should be placed on the top of the class or interface declaration. A block of code containing the code before the class documentation and until a new doc-comment block appears will be converted to a LaTeX code block. Any subsequent block of comments will be inserted as LaTeX contents after the declaration or body following the documentation. A declaration is detected by the presence of a semicolon (;) on a line. An end of body is detected when a balanced set of opening and closing brackets is found.

The tab characters will be replaced by individual spaces, in order to be correctly displayed by LaTeX.

5 Running Javatex

As for `texjava.pl`, the script requires a Perl 5 interpreter to be installed but LaTeX`2HTML` is not necessary for `javatex.pl`. The following command runs the script:

`perl javatex.pl` [`-tabsize` *i*] *infile outfile*

Here, `javatex.pl` has to be replaced by a path to the script if it is not executed in the `tcode` directory. Its arguments and option are as follows:

infile

The name of the input file, which should have the `.java` extension. The input file is parsed but not modified.

outfile

The name of the output file, which should have the `.tex` extension. If *outfile* already exists, it will be replaced without notice, otherwise the file will be created in the working directory. If *infile* and *outfile* are the same, an error will occur.

`-tabsize` *i*

By using the `-tabsize` *i* option, where *i* is a non-negative integer, one can specify the number of spaces that corresponds to a tab character. The default value is 8.

Example: To extract the LaTeX code from the Java file `Chrono.java` and place it in file `Chrono.tex`, one can use:

```
perl javatex.pl  Chrono.java  Chrono.tex
```

6 Ant tasks

Apache Ant is becoming the de facto build system for Java applications. An XML build file instructs Ant how to build the Java package in a portable way. Unfortunately, Ant is not a scripting language and can only deal with plain `.java` file. It could be possible to call `texjava.pl` manually using the Ant built-in `exec` task, but this would fastly because tedious and no dependency checking would be performed at all. On each build cycle, every `.tex` file would be turned into a `.java` and recompiled, since it would be freshly created, into a new `.class` file. However, it is possible to extend Ant using user-defined tasks. TCode provides some Ant tasks to work with the LaTeX files. The `texjava` task allows one to process a batch of `.tex` files into `texjava.pl`, producing output `.java` files. The `pdflate` task allows one to construct, using pdfLaTeX, the PDF documentation for a package. To use these tasks, one must include `tcode.jar` in the CLASSPATH environment variable and declare the use tasks in the `build.xml` file, using the `taskdef` task. For example, to use `texjava` task, one must write, in the `project` section of `build.xml`,

```
<taskdef name="texjava" classname=''umontreal.iro.lecuyer.tcode.Texjava"/>
```

Texjava

This Ant task invokes `texjava.pl` to extract Java code from LaTeX files. Given a list of files, it can invoke `texjava.pl` once for each, creating Java file that can be processed with Ant. The `Texjava` task checks for depencies, avoiding to run `texjava.pl` for `.tex` files with a more recent `.java` file. It acts like a proxy to `texjava.pl` so all of its options are supported by the task through attributes. It is possible to set default values for most of these attributes using system properties. See the documentation of `texjava.pl` for more detailed information about the options.

Note: For this class to work, the Perl interpreter must be in the PATH environment variable.

Supported system properties The system properties allows one to set default attributes common to all `texjava` task usage. It reduces the tedium of writing the `texjava` calls and provides an easier way to modify parameters later.

`texjava.html`

`texjava.images`

`texjava.savelatex`

`texjava.htmloutdir`

`texjava.texjava`

This property specifies the path to the `texjava.pl` file, including the name of the script. Since no assumption can be made about the current directory, it is recommended to give an absolute path or a path relative to one's project base directory. If the path is not specified, `tcode/texjava.pl` will be assumed. This imply that one's project will have a `tcode` directory containing `texjava.pl`.

Available attributes. Attributes are use to customize a single call to the `texjava` task. They have precedence on system properties.

`html`

`images`

`savelatex`

`htmlonly`

`htmloutdir`

`master`

`texjava`

This specifies the path to the `texjava.pl` script.

`overviewtopackage`

Normally, any `.tex` file is mapped to a corresponding `.java` for dependency checking and conversion. If `htmlonly` is `true`, `.tex` files will be mapped to `.html` files. This attribute, if set to `true`, will define a mapping from `overview.tex` to `package.html`. This allows one to generate package overviews without converting them every time.

Nested elements Two nested elements are supported: file sets and file lists. These elements allows one to construct the list of `.tex` files to be converted to `.java` files.

`texfileset`

Corresponds to an Ant `FileSet` element which should only contain `.tex` files. A set can include files matching patterns.

`texfilelist`

Corresponds to an Ant `FileList` element which should only contain `.tex` files. When making a file list, one must specify the names of each individual files.

Example. For example, the compile target of Probdist uses the task to create the Java files and compiles them.

```
<target name="probdist"
    description="Compiles probability distribution package">
   <texjava master="src/${pprobdist}/guideprobdist.tex">
      <texfilelist dir="src/${pprobdist}"
         files="DiscreteDistribution.tex,ContinuousDistribution.tex"/>
      <texfileset dir="src/${pprobdist}" includes="*Dist.tex"/>
   </texjava>
   <javac srcdir="src" destdir="build" includes="${pprobdist}/*.java"/>
   <texjava overviewtopackage="yes" html="yes" htmlonly="yes"
              master=’’src/${pprobdist}/guideprobdist.tex">
      <texfilelist dir="src/${pprobdist}" files="overview.tex"/>
   </texjava>
</target>
```

Here, the `pprobdist` system property refers to the location of the Probdist package, i.e., `umontreal/iro/lecuyer/probdist`. This will generate only the needed Java file and one is free to add classes ending with the `Dist` suffix without modifying the build file.

```
package umontreal.iro.lecuyer.tcode;
```

```
public class Texjava extends Task
```

`public boolean getHtml()`

Returns `true` if the task generates HTML contents.

`public void setHtml (boolean html)`

If set to `true`, HTML contents will be generated.

`public boolean getImages()`

Returns `true` if the image generation is activated.

`public void setImages (boolean images)`

If set to `true`, LaTeX`2HTML` will generate images when converting the documentation.

`public boolean getSavelatex()`

Returns `true` if the LaTeX contents is saved to allow reverting to LaTeX using *javatex*.

`public void setSavelatex (boolean savelatex)`

If set to `true` the LaTeX contents will be saved in the produced Java file.

`public boolean getHtmlonly()`

Returns `true` if only HTML code is to be produced.

`public void setHtmlonly (boolean htmlonly)`

If set to `true`, only HTML code will be produced. No Java code and comments will appear.

`public boolean getOverviewtopackage()`

Returns `true` if `package.html` will be created from `overview.tex`.

`public void setOverviewtopackage (boolean pack)`

If set to `true`, a file named `overview.tex` will be mapped to `package.html` instead of `overview.html`.

`public File getMaster()`

Gets the name of the master file.

`public void setMaster (File masterFile)`

Sets the name of the master file.

`public File getHtmloutdir()`

Gets the directory where the HTML files will be created.

`public void setHtmloutdir (File htmloutdir)`

Sets the directory where the HTML files will be created.

`public String getHtmltitle()`

Gets the title of the HTML page.

`public void setHtmltitle (String title)`

Sets the title of the HTML page.

`public File getTexjava()`

Returns the path to `texjava.pl`.

`public void setTexjava (File texjava)`

Sets the path to `texjava.pl`.

`public void addTexfileset (FileSet fs)`

Adds a set of `.tex` files to be converted to `.java` files.

`public void addTexfilelist (FileList fl)`

Adds a list of `.tex` files to be converted to `.java` files.

`public void execute() throws BuildException`

Executes the conversion task, using the Ant `exec` task to invoke `texjava.pl` on each `.tex` file.

PdfLatex

This task allows one to automate the PDF documentation generation. It uses pdfLaTeX to convert the LaTeX documentation into PDF. To avoid errors when using the LaTeX `hyperref` package, all `.aux` files are first deleted. The pdfLaTeX program is then called one time to make an initial run. BIBTeX is called to generate to bibliography file, then pdfLaTeX is called two more times to fix references and table of contents. It always runs pdfLaTeX in *nonstop* interaction mode, preventing it from blocking on errors. It is therefore recommended to sometimes run pdfLaTeX manually to ensure that all documentation is free from syntax errors. This seems a lot of work, but it is necessary to get a fully-automated documentation building system.

Supported attributes. This task supports only one attribute.

`latexfile`
This gives the name of the file to be converted.

Example. As an example, this is the command used to produce the PDF file for the `util` package.

```
<pdflatex latexfile="src/umontreal/iro/lecuyer/util/guideutil.tex"/>
```

```
package umontreal.iro.lecuyer.tcode;

public class PdfLatex extends Task
```

`public File getLatexfile()`
Returns the name of the LaTeX file to be processed.

`public void setLatexfile (File latexFile)`
Sets the name of the LaTeX file to be processed.

`public void execute() throws BuildException`
Executes the task.

References

[1] ActiveState. *ASPN: Reference—ActivePerl Docs*, 2003. Available online at `http://aspn.activestate.com/ASPN/Reference/Products/ASPNTOC-ACTIVEPERL`.

[2] D. Flanagan. *Java in a Nutshell*. O'Reilly, Sebastopol, CA, third edition, 1999.

[3] M. Goossens and S. Rahtz. *The LaTeX Web Companion*. Addison-Wesley, 1999.

[4] D. E. Knuth and S. Levy. *The CWEB System of Structured Documentation*. Addison-Wesley, Reading, MA, 1994.