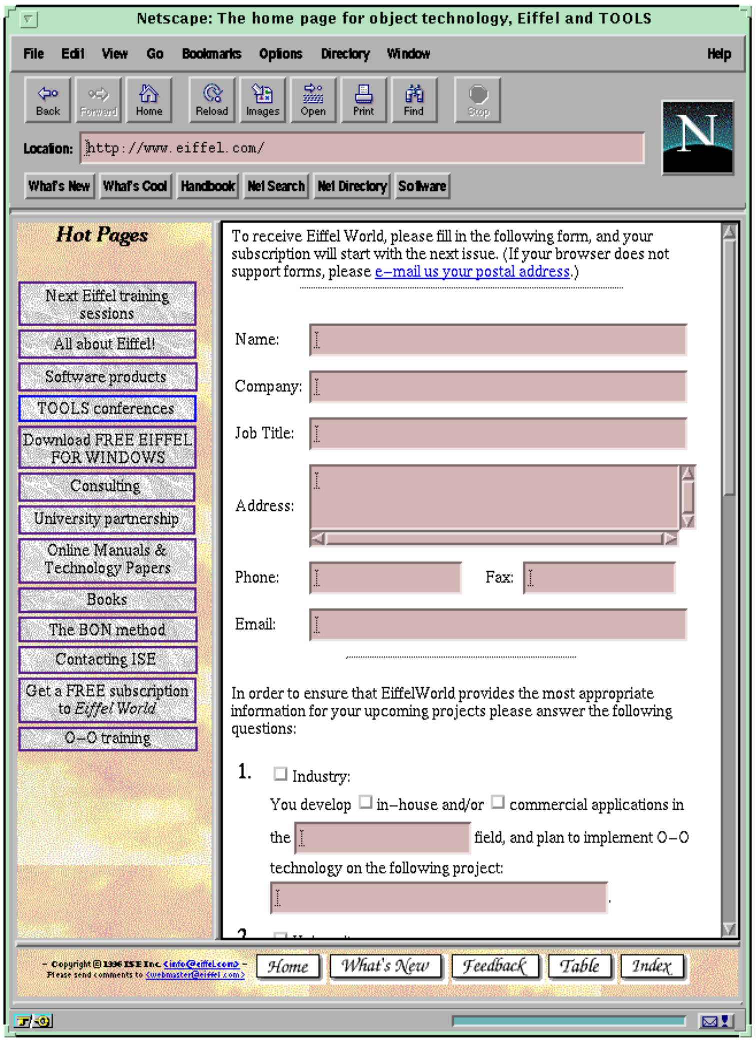
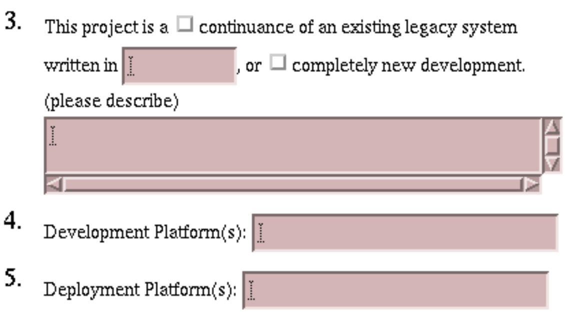


EiffelWeb:

Processing forms from the Web using Eiffel



Report identification

EiffelWeb: Processing forms from the Web with Eiffel, ISE Technical Report TR-EI-60/WE.

Publication history

First published: August 1996.

Version: 3.3.9, August 1996. Corresponds to version 3.3.9 of the ISE Eiffel 3 environment.

Authors

Fabrice Franceschi, Bertrand Meyer

Reviewers: Xavier Le Vourch, Moriah Sandberg.

Software credits

EiffelWeb was designed and implemented by Fabrice Franceschi. Project leader: Xavier Le Vourch.

Cover design

ISE.

CONTENTS

1 THE ROLE OF EIFFELWEB

The World-Wide Web provides a remarkably versatile way to make information available to the world. But the flow should not just be one way: a Web site will want to obtain input from its visitors. This can provide feedback, new ideas, and sales leads.

The simplest way for a Web site visitor to send such input is to fill in a **form** on one of the site's pages. This requires, on the other end, some software to process the form and extract its information. Such software is known as a **CGI script**, where the initials stand for Common Gateway Interface.

Before EiffelWeb, CGI scripts were usually written in scripting languages such as Perl, or in C. For very simple processing of small forms this suffices; but scripts, like any other software,must grow and evolve. Object technology can offer a better solution — easier to produce, easier to maintain, easier to extend — than low-level approaches.

The EiffelWeb library provides script developers with a simple set of tools to develop CGI scripts using the full benefits of Eiffel.

Why use EiffelWeb?

Although the concepts should be simple, form processing can be unpleasantly tedious. EiffelWeb provides a general framework, taking care of all the repetitive details, and frees you of many low-level tasks. In particular:

- You do not need to worry about which of the two available methods (**GET** and **POST**) has been used to return data. One uses an environment variable, the other writes to the standard input, but EiffelWeb will get the data, wherever it is, to your application.
- The basic format of the data returned from a form is very low-level: a set of fields separated by **&** characters, with special encoding for blanks and non-alphabetic characters. With ordinary CGI scripts, you must parse this information. EiffelWeb does the parsing for you and makes the resulting fields directly available.
- After processing a form you will often display further information to the person who has submitted it. This information must be in HTML format (HTML, HyperText Markup Language, is the language used to format documents to be displayed by Web browsers). With ordinary scripting languages, you must take care of the HTML codes yourself. With EiffelWeb, you can use a set of high-level procedures taking care of the basic forms of output; the procedures will generate the proper HTML codes for you. You do not need to know HTML to use EiffelWeb.
- Debugging a form-processing system can be extremely tricky. EiffelWeb provides you with a rich set of mechanisms to try out your application, set environment variables, specify what should happen in the case of an exception.
- Best of all, the actual processing can benefit from the whole power of the Eiffel language, the ISE Eiffel environment, and the ISE Eiffel libraries. You have a full development

environment at your disposal, rather than the limited expressive power of a simple scripting language.

Organization of this document

This manual shows how to use EiffelWeb. It is organized as follows:

- Below is a short description of the EiffelWeb classes.
- Chapter 2 shows how to write a small form-processing application. The example will be sufficient for many simple uses.
- Chapter 3 gives some background about HTML form processing.
- Chapter 4 explains how your application can access user information and environment variables.
- Chapter 5 describes debugging facilities.
- Chapter 6 presents the class *HTML_GENERATOR* for generating HTML code.
- Appendix A presents the principal class, *CGI_INTERFACE.*.
- Appendix B presents class *CGI_ENVIRONMENT.*
- Appendix C presents class *HTML_GENERATOR.*
- Appendix D presents class *HTML_CONSTANTS*, used by *HTML_GENERATOR*.

Classes

Here is a short description of each class:

- *CGI_ENVIRONMENT*: provides access to environment variables, including all the standard variables.
- *CGI_INTERFACE*: a deferred class that encapsulates the behavior of all CGI scripts. It includes the data processing operations common to all scripts, namely the parsing of the string sent by the server and the debugging facilities.
- *HTML_CONSTANTS*: defines a subset of the HTML language tags as Eiffel constants. Its obvious purpose is to be used for HTML generation (more will be added in future versions).
- *HTML_GENERATOR*: implements a series of routines facilitating the generation of basic HTML documents. The output medium can be changed by simply redefining the routine *put_basic*.

2 WRITING A FORM-PROCESSING SYSTEM

Processing Web formd with EiffelWeb is straightforward, as EiffelWeb takes care of parsing the form and makes the visitor's input available to you in the form of Eiffel objects, instances of EiffelWeb classes and their descendants.

Writing a basic script

Here are the basic steps to process the data entered by someone (called the "*visitor*" in the rest of this document) who has filled a form on your Web site:

- Write an Eiffel class *YOUR_FORM* describing your form. *YOUR_FORM* should be a descendant from the EiffelWeb class *CGI_INTERFACE*, itself an heir to *CGI_ENVIRONMENT* which provides access to environment variables.
- Write in *YOUR_FORM* an effective version of the deferred feature *execute* inherited from *CGI_INTERFACE*. This is the body of your form processing, where you can manipulate the data entered by the visitor and, if desired, produce HTML output for display to the visitor.
- Either define the *make* creation procedure inherited from *CGI_INTERFACE* as a creation procedure of *YOUR_FORM*, or write your own creation procedure if you need more specific creation actions. The default *make* starts off everything as needed and calls *execute*; for most uses this is what you need.

A complete example

Let us build a small example. Our system will simply display the results of any HTML form as an indented list of name-value pairs. The name of the field will appear in bold face.

You only need one class, the one called *YOUR_FORM* above — here *CGI_DEMO*; it can keep the inherited *make* as a creation procedure. Here is the class:

```
class
    CGI_DEMO
inherit
    CGI_INTERFACE
creation
    make
feature
    execute is
            -- Analyze form's fields and output them as name-value pairs.
        do
            ... See next ...
        end
end -- class CGI_DEMO
```

with *execute* effected as follows:

```
execute is
        -- Analyze form's fields and output them as name-value pairs.
    local
        i: INTEGER;
        vl: LINKED_LIST [STRING]
    do
        generate_html_header;
        put_basic ("You have submitted the  following name-value pairs:");
        put_glossary_start
        from
            i := fields.lower
        until
            i > fields.upper
        loop
            put_glossary_term (fields.item (i));
            put_glossary_definition;
            put_line_break;
            vl := value_list (fields.item (i));
            from
                vl.start;
            until
                vl.after
            loop
                put_basic (vl.item);
                vl.forth;
                if not vl.after then put_line_break end
            end;
            i := i + 1
        end
        put_glossary_end
    end
end -- class CGI_DEMO
```

Procedure *execute* loops over the elements of the array *fields*, which holds the names of the fields. For each field it loops over the list of values for that field, which may have zero, one or more elements. Procedure *generate_html_header*, which like the other features used by this version of *execute* comes from *CGI_INTERFACE*, prints on the standard output the header of an HTML reply, which is necessary to avoid getting an error from the Web server.

You will have noted that this text does not contain any HTML code. This is because using EiffelWeb does not require knowing HTML; you can produce output using abstract output procedures such as *put_line_break*, *put_basic* and *generate_html_header* which will take care of generating the appropriate HTML output. If you do know HTML and want to generate specific output, you can do so using the class *HTML_GENERATOR* (see section 6).

3 HOW DO FORMS GET PROCESSED?

To help understand EiffelWeb it is useful to have a general picture of the tasks involved in processing a form received through the World-Wide Web.

When a visitor clicks on the *submit* button of an HTML form, the visitor's Web browser sends the contents of text fields, selected buttons and items to the server. In turn, the server transmits these data to the CGI script associated with the form. The transfer can be done via two distinct methods:

- **GET**: data is made available through the environment variable, *QUERY_STRING*
- **POST**: data is available on the standard input of the CGI script

With either method, all the fields are concatenated into a unique string which has the following structure:

*field1=value1***&***field2=value2***&***...*

Space characters are converted to '+' and any special character is replaced by its hexadecimal code.

Execution of the CGI script also sets a number of environment variables, defining: the method, **GET POST**, with which the script was called (*REQUEST_METHOD*); the length of the data (*CONTENT_LENGTH*); the name of the server (*SERVER_NAME*); and others.

A CGI script written in an ordinary scripting language will typically perform the following tasks in response to a visitor submitting a form:

- Determine the method with which it was called, so as to know where to get the input. With EiffelWeb you will not need this step, as EiffelWeb takes care of finding the input for you.
- Parsing the input, to decode the fields and obtain their values. With EiffelWeb you do not need to worry about this aspect, as EiffelWeb decodes everything and puts the result into Eiffel objects.
- Process the input as needed by the application. EiffelWeb provides a specific place (the procedure *execute* of class *CGI_INTERFACE)* for such processing, various features and an array *fields* containing all the input fields, and of course the whole power of ISE Eiffel and its libraries to perform all the processing that you need.
- Most of the time, produce output to be sent to the visitor in HTML form. Here too EiffelWeb provides all the necessary output facilities, taking care of generating the proper HTML syntax for you.

4 ACCESSING VISITOR INFORMATION AND ENVIRONMENT VARIABLES

EiffelWeb provides a set of classes to access information provided by the visitor, as well as the values of various environment variables.

Visitor information

When a visitor enters information into a form, the Web server gets it in a raw, hardly usable form. This is where EiffelWeb steps in: it parses the information, and makes it available in a form that is convenient for direct processing by your classes according to your needs.

A given field may have zero, one or move values, depending both on the field's type and on what the visitor entered in it. In usual CGI processing, there is no easy way to find out beforehand. EiffelWeb, however, enables you to obtain the information through a number of queries from class *CGI_INTERFACE*, which you can all apply to a field:

- The query *value_count* yields the number of values available for a field; this is particularly useful when you need to process multiple values differently from single values
- In the single-value case, the query *value* yields that value. (For multiple values it yields the first one.)
- In the multiple-value case, the query *value_list* returns a *LINKED_LIST*containing the values.

In the case of checkboxes and similar kinds of fields, your system will not see any field, not even an empty one, if the user did not check the box. To find out whether a field is available, use the query *field_defined*.

If, as is often the case, you know the name of a field, you can just pass that name as a string argument to any of the preceding queries. You can also, as in the example of section 1, loop over all the fields of the form, using the array

fields: ARRAY[STRING]

and passing the value of *fields*•*item (i)*, for various *i*, to the the above queries:

All the features listed in this section belong to class *CGI_INTERFACE*, whose complete specification appears in Appendix A.

Accessing environment variables

A Web server can set a number of environment variables, which will not change during the processing of a form and so can be represented as once functions, all returning strings.

These strings are available from features of class *CGI_ENVIRONMENT*, whose complete specification appears in Appendix B. *CGI_ENVIRONMENT* also includes a procedure *set_ environment_variable* to set environment variables to specific values, useful only for debugging.

5 USING THE DEBUGGING FACILITIES

One of the most difficult steps in writing form processing software with traditional CGI scripts is debugging. When the script does not complete successfully, it exits with an error code; in such a case the Web server merely reports an internal error occurred, which is not much help.

EiffelWeb makes the debugging process much more convenient through features of class *CGI_INTERFACE*.

Using *make_debug* as a creation procedure

Besides *make*, *CGI_INTERFACE* provides another procedure that can be used as creation procedure: *make_debug*, which extends *make*. Using *make_debug* you can perform extra operations, for example set environment variables to special values, and you can parse the input before starting any processing.

A piece of methodological advice: once you are done with debugging, remember to set the creation procedure back to the normal *make*, just in case *make_debug* sets environment variables to values which are not appropriate for normal processing, or generates output that is intended for you and not for your visitors.

Environment variables

To recreate the context in which the application is called by the WWW server, you need to set the corresponding environment variables. This can lead to one of the most annoying aspects of traditional CGI script creation and debugging. You must exit the working environment, set the variables, and restart the environment; or you could set the variables from the script, but this is just as inconvenient.

EiffelWeb provides a better solution: set the variables on-the-fly by passing them as argument to the script. You will for example call the application as:

your_script REQUEST_METHOD=GET QUERY_STRING="*name=foo*"

setting the variables *REQUEST_METHOD* and *QUERY_STRING* to *GET* and to *name=foo*.

To set the default values, *make_debug* calls the procedure *set_environment*; in its version inherited from *CGI_INTERFACE*, *set_environment* does nothing, but you can redefine it to set environment variables to specific values. To set an individual environment variable, use *set_environment_variable*. The defaults will be overriden by the scripts arguments if any.

Here is an example class using these facilities:

```
class
    CGI_SCRIPT
inherit
    CGI_INTERFACE
        redefine
            set_environment
        end
creation
    make_debug
feature
    set_environment is
            -- Set two environment variables.
        do
            set_environment_variable ("SCRIPT_NAME", "form_handler");
            set_environment_variable ("SERVER_NAME", "test.oursite.com");
        end
end -- class CGI_SCRIPT
```

Of course, you can still test your script manually and set the environment variables from the command line as follows (here expressed in C-shell syntax):

```
setenv REQUEST_METHOD "GET"
setenv QUERY_STRING ""
setenv CONTENT_LENGTH 0
setenv SERVER_NAME "no_server"
setenv SCRIPT_NAME "test_script"
```

Error handling

The creation procedure, whether *make* or *make_debug*, will catch any exception occurring during the execution of the script and will handle it according to the option that you will have selected by calling (normally in *set_environment*) one of the following three procedures:

- *set_no_debug*: an exception will terminate processing with no further action. This is the default.
- *set_message*: an exception will terminate processing and displays a message explaining that an internal error occurred.
- *set_exception_trace*: an exception will cause the the script to terminate and display the exception trace. This will only be in effect if the Lace option *exception_trace* is on; this is the case by default in all modes except final mode, for which you may turn it on through the line *exception_trace* (**yes**) in the **default** paragraph of your Ace.

In all cases the application will not return an error code but keep control of the execution, therefore preventing the WEb server from displaying its own message.

If you detect an irrecoverable error during processing, you can stop execution using procedure *raise_error*. This will display an error message, which you can set through an earlier call to *set_error*.

All the features listed in this section belong to class *CGI_INTERFACE*, whose complete specification appears in Appendix A.

An example using *make_debug*

We may use the basic example again to illustrate the debugging mechanism. Using *make_debug* instead of *make* as the creation procedure will allow us to change the debug level - e.g. to get the exception trace if an exception occurs. This will also allow us to set default values for some environment variables. This time instead of an indented list the results will be displayed in a bulleted list.

```
class
    CGI_DEMO

inherit
    CGI_INTERFACE
        redefine
            set_environment
        end;
    HTML_GENERATOR

creation
    make_debug

feature

    set_environment is
        do
            set_exception_trace;
            set_environment_variable ("SERVER_NAME", "no_server");
            set_environment_variable ("SCRIPT_NAME", "test_script")
        end

    execute is
        local
            i: INTEGER;
            vl: LINKED_LIST[STRING]
        do
            Generate_html_header;
            put_basic ("You have submitted the following name / value pairs:");
            put_unordered_list_start;
```

```
            from
                i := fields.lower
            until
                i > fields.upper
            loop
                put_list_item_start;
                put_bold (fields.item (i));
                put_line_break;
                vl := value_list (fields.item (i));
                from
                    vl.start
                until
                    vl.after
                loop
                    put_basic (vl.item);
                    vl.forth;
                    if not vl.after then
                        io.putstring (", ")
                    end
                end;
                put_list_item_end;
                i := i + 1
            end;
            put_unordered_list_end
        end
end -- class CGI_DEMO
```

6 GENERATING HTML OUTPUT

Many form-processing systems will have to display information back to the visitor, if only to thank the visitor for filling in the form.

This information must be in HTML format; it will often include some of the form's input, or elements generated from that input.

To avoid forcing you to write low-level HTML code, the class *HTML_GENERATOR* provides a set of procedures to take care of the most common cases, such as:

- *put_bold*: write a string in boldface.
- *put_header1*: write a level 1 title.
- *put_preformatted*: write a string exactly as it was.

You can still, of course, include specific HTML codes if you wish. The procedures provided cover only a subset of HTML.

By default all written information will go to the standard output. You can redirect it through the procedure *put_basic*.

All the features listed are part of class *HTML_GENERATOR* whose complete specification appears in Appendix C.

A

Class *CGI_INTERFACE*

indexing

description: "*Access to information provided by a user through an HTML form. This class may be used as ancestor by classes needing its facilities*."
status: ""
date: "*$Date*: *$*"
revision: "*$Revision*: *$*"

deferred class interface
CGI_INTERFACE

feature -- Initialization

make
-- Initiate input data parsing and process information.

make_debug (*args*: *ARRAY* [*STRING*])
-- Set environment variables and proceed to regular execution.

feature -- Access

fields: *ARRAY* [*STRING*]
-- Names of fields in the form.

value (*field_name*: *STRING*): *STRING*
-- First (unique?) value for a field.
require
field_not_void: *field_name /= Void*;
field_exists: *field_defined* (*field_name*)
ensure
value_exists: *Result /= Void*

```
value_count (field_name: STRING): INTEGER
        -- Number of values for a field.
    require
        field_not_void: field_name /= Void;
        field_exists: field_defined (field_name)

    ensure
        valid_count: Result >= 0

value_list (field_name: STRING): LINKED_LIST [STRING]
        -- List of values for a field.
    require
        field_not_void: field_name /= Void;
        field_exists: field_defined (field_name)
    ensure
        valid_count: Result.count = value_count (field_name)

feature -- Status report

    field_defined (field_name: STRING): BOOLEAN
            -- Is field field_name defined?

feature -- Status setting
    set_exception_trace
            -- Cause any future exception to display exception trace.

    set_message
            --Cause any future exception to display error message.

    set_no_debug
            -- Cause any future exception to do nothing.

feature -- HTTP facilities

    generate_html_header
            -- Generate CGI header reply.

feature -- Miscellanous
    execute
            -- Process user-provided information.

    set_environment
            -- Set environment variable to user value.

end -- class CGI_INTERFACE
```

B

Class *CGI_ENVIRONMENT*

indexing

description: "*Access to environment variables set by the HTTP server when the CGI application is executed. This class may be used as ancestor by classes needing its facilities.*"
status: ""
date: "*$Date*: *$*"
revision: "*$Revision*: *$*"

class interface
CGI_ENVIRONMENT

feature -- Environment variable setting
set_environment_variable (*variable*, *val*: *STRING*)
-- Set environment variable *variable* to *val*.
require
valid_variable: *variable /= Void and then variable.count* > *0*;
valid_value: *val /= Void*

feature -- Non-request-specific environment variables
Gateway_interface: *STRING*
-- Revision of the CGI specification to which this server complies

Server_name: *STRING*
-- Server's hostname, DNS alias, or IP address

Server_software: *STRING*
-- Name and version of information server answering the request

feature -- Headerline based environment variables

Http_accept: *STRING*
-- MIME types which the client will accept

```
Http_user_agent: STRING
        -- Browser the client is using to send the request

feature -- Request-specific environment variables
    Auth_type: STRING
            -- Protocol-specific authentication method used to validate user

    Content_length: STRING
            -- Length of the said content as given by the client

    Content_type: STRING
            -- Content type of data

    Path_info: STRING
            -- Extra path information, as given by the client

    Path_translated: STRING
            -- Translated version of PATH_INFO provided by server

    Query_string: STRING
            -- Information following  ? in URL referencing CGI program

    Remote_addr: STRING
            -- IP address of the remote host making the request

    Remote_host: STRING
            -- Hostname making the request

    Remote_ident: STRING
            -- User name retrieved from server if RFC 931 supported

    Remote_user: STRING
            -- User name, if applicable

    Request_method: STRING
            -- Method with which the request was made

    Script_name: STRING
            -- Virtual path to the script being executed

    Server_port: STRING
            -- Port number to which request was sent

    Server_protocol: STRING
            -- Name and revision of information protcol of this request
end -- class CGI_ENVIRONMENT
```

C

Class *HTML_GENERATOR*

indexing

description: "*HTML generation. This class may be used as ancestor by classes needing its facilities*"
status: ""
date: "*$Date: $*"
revision: "*$Revision: $*"

class interface
HTML_GENERATOR

feature -- Transformation

put_basic (*s*: *STRING*)
-- Write *s* to medium.
require
string_not_void: *s* /= *Void*

feature -- Miscellaneous

put_bold (*text*: *STRING*)
-- Write *text* in bold face.
require
text_not_void_void: *text* /= *Void*

put_glossary_definition
-- Write glossary definition tag.

put_glossary_end
-- End glossary list.

put_glossary_start
-- Start glossary list.

put_glossary_term (*text*: *STRING*)

-- Write *text* as glossary term.

require

text_not_void: *text* /= *Void*

put_header1 (*title*: *STRING*)
-- Write *title* as level 1 header.
require
title_not_void: *title* /= *Void*

put_header2 (*title*: *STRING*)
-- Write *title* as level 2 header.
require
title_not_void: *title* /= *Void*

put_header3 (*title*: *STRING*)
-- Write *title* as level 3 header.
require
title_not_void: *title* /= *Void*

put_header4 (*title*: *STRING*)
-- Write *title* as level 4 header.
require
title_not_void: *title* /= *Void*

put_header5 (*title*: *STRING*)
-- Write *title* as level 5 header.
require
title_not_void: *title* /= *Void*

put_header6 (*title*: *STRING*)
-- Write *title* as level 6 header.
require
title_not_void: *title* /= *Void*

put_horizontal_rule
-- Write an horizontal rule.

put_italic (*text*: *STRING*)
-- Write *text* in italic.
require
text_not_void: *text* /= *Void*

put_line_break
  -- Write line break.

put_link (*url*, *anchor*: *STRING*)
  -- Attach text *anchor* to *url*.
 require
  anchor_not_void: *anchor* /= *Void*;
  url_not_void: *url* /= *Void*

put_list_item_end
  -- Write list item end tag.

put_list_item_start
  -- Write list item start tag.

put_ordered_list_end
  -- End ordered list.

put_ordered_list_start
  -- Start ordered list.

put_paragraph_end
  -- End paragraph

put_paragraph_start
  -- Start paragraph.

put_preformatted (*text*: *STRING*)
  -- Write *text* as preformatted text.
 require
  text_not_void: *text* /= *void*

put_unordered_list_end
  -- End unordered list.

put_unordered_list_start
  -- Start unordered list.

end -- class *HTML_GENERATOR*

D

Class HTML_CONSTANTS

indexing

description: "*Tags of subset of the HTML language. This class may be used as ancestor by classes needing its facilities*"
status: ""
date: "*$Date*: *$*"
revision: "*$Revision*: *$*"

class interface
HTML_CONSTANTS

feature -- Constants

Bold_end: *STRING is* "*</B>*"
-- Bold face end tag

Bold_start: *STRING is* "*<B>*"
-- Bold face start tag

Glossary_definition: *STRING is* "*<DD>*"
-- Glossary list definition

Glossary_end: *STRING is* "*</DL>*"
-- Glossary list end tag

Glossary_start: *STRING is* "*<DL>*"
-- Glossary list start tag

Glossary_term: *STRING is* "*<DT>*"
-- Glossary list term

H1_end: *STRING is* "*</H1>*"
-- Level 1 header end tag

H1_start: *STRING is* "*<H1>*"
-- Level 1 header start tag

H2_end: *STRING is* "</H2>"
-- Level 2 header end tag

H2_start: *STRING is* "<H2>"
-- Level 2 header start tag

H3_end: *STRING is* "</H3>"
-- Level 3 header end tag

H3_start: *STRING is* "<H3>"
-- Level 3 header start tag

H4_end: *STRING is* "</H4>"
-- Level 4 header end tag

H4_start: *STRING is* "<H4>"
-- Level 4 start tag

H5_end: *STRING is* "</H5>"
-- Level 5 header end tag

H5_start: *STRING is* "<H5>"
-- Level 5 header start tag

H6_end: *STRING is* "</H6>"
-- Level 6 header end tag

H6_start: *STRING is* "<H6>"
-- Level 6 header start tag

Horizontal_rule: *STRING is* "<HR>"
-- Horizontal rule tag

Italic_end: *STRING is* "</I>"
-- Italic end tag

Italic_start: *STRING is* "<I>"
-- Italic start tag

Line_break: *STRING is* "
"
-- Line break tag

List_item_end: *STRING is* "</LI>"
-- List item end tag

List_item_start: *STRING is* "*<LI>*"
-- List item start tag

Ordered_list_end: *STRING is* "*</OL>*"
-- Ordered list end tag

Ordered_list_start: *STRING is* "*<OL>*"
-- Ordered list start tag

Paragraph_end: *STRING is* "*</P>*"
-- Paragraph end tag

Paragraph_start: *STRING is* "*<P>*"
-- Paragraph start tag

Preformatted_end: *STRING is* "*</PRE>*"
-- Preformatted text end tag

Preformatted_start: *STRING is* "*<PRE>*"
-- Preformatted text start tag

Unordered_list_end: *STRING is* "*</UL>*"
-- Unordered list end tag

Unordered_list_start: *STRING is* "*<UL>*"
-- Unordered list start tag

end -- class *HTML_CONSTANTS*