



Exclamation!

Desktop Web Publisher

Reference Manual

October 18, 2003

Ten League Boots™



Copyright © 2003 Ten League Boots. All rights reserved.

Comments and feedback welcome. Please e-mail support@tenleagueboots.com



Contents

Introduction	7
Using the manual	7
Exclamation—a brief overview	7
Content and the content database	9
Database	9
Types	10
Formats	11
Using more than one database	11
Content Declaration file	13
XML Syntax	14
<content>	15
<tabdatabase>	15
<folder>	15
<table>	17
<column>	17
<subtable>	18
Data types for <column> data	19
type=date[: <i>format</i>]	19
type=decimal[: <i>precision</i>]	22
type=html	22
type=integer	23
type=paragraph	23
type=plaintext	23

type=safetext	23
Template Declaration files	25
XML Syntax	27
<template>	27
<input file="filepath" />	27
<output file="filepath" "filepathexpression" />	27
<expression name="exprname"> expression </expression>	29
<query table="tablename" sortby="columnname asc desc,..." ></query>	29
<keep> selection expression </keep>	30
<omit> selection expression </omit>	30
<segmentlist name="listname" bycount="integer" byequal="colname" >	31
<rowlist name="listname"/>	32
Subtable queries	32
Template declaration examples	34
Linking queries	35
About circular definitions	37
HTML template files	39
Substitution	39
Exclamation HTML extensions	41
Exclamation "commands" in HTML tags	41
Exclamation TLB tag	41
Conditional commands	43
Looping commands	46
Implicit looping	49
Common Syntax	55
File naming conventions	55
HTML 4.01 review	55
Begin and end tagged elements	56
Begin-only tagged elements	56
XML syntax	57

⋮

Exclamation syntax	58
Symbols	58
Constants	58
Expressions	59
Decimal math	66
Substitution formatting	68
General syntax	68
Number formatting	68
Date and time formatting	72
Character set conversions	77
How character sets are determined by Exclamation	77
Character set names	78
Our recommendation	79
Index	81

·
·
·



Introduction

The Exclamation Reference Manual is intended for Web designers and developers creating a Site Design for Exclamation Publisher. It provides reference material only—lists and examples arranged for easy lookup. This manual assumes that you have a basic knowledge of Exclamation and how it works. It therefore assumes you have already read the Exclamation Designer's Guide.

Using the manual

It is much easier to understand syntax once one has a basic grasp of the language. Exclamation syntax is fairly intuitive, and furthermore many readers will never need more than a basic understanding. For these reasons, the Syntax chapter is last. The first and second chapters discuss Content Declaration and Templates respectively.



For ease of reference, as far as possible all the material for a particular subject is in one place. However, because Exclamation is both simple to use and sophisticated, there are two basic types of material: that which *all* readers need to know, and that which only advanced users need. The advanced material is marked with the wizard icon in the margin. The wizard may be against a paragraph or a section head. Against a paragraph means just this paragraph is advanced, and against a section means the whole section is advanced. Note that like the syntax material, it is entirely possible to use Exclamation's power for creating and maintaining Web sites and *never* need the wizard's sections. Feel free to just skip over them.

Exclamation—a brief overview

Exclamation Publisher is a suite of tools for the creation and maintenance of small to medium-sized (10 to 5,000 pages) *content-driven* Web sites. It completely separates site design from content creation and maintenance. Exclamation exploits this feature by allowing Web designers to provide a simple interface to client's content, which the client can then use to



keep the content up-to-date. And since regenerating the site is a matter of clicking on the buttons—without touching (and potentially breaking) the HTML code—the client can easily and safely update the site as frequently as she wishes.

Thus Exclamation allows the design to be changed without re-entering the content, and the content to change without the need to edit the code. Together, this makes the site vital, attractive and never stale.

The desktop content editor is customized to the client's content by the developer and is one of the deliverables. Exclamation makes it possible to use common desktop applications for this purpose.

The business manager and Web designer need to decide what parts of a page will be editable. The general rule is that the editable material goes into the content database, and the rest of the site is statically coded in HTML. There will be give and take here, since the final decision may depend on additional considerations.

Exclamation maintains a *project* file which it uses to record relevant details such as where the content, template, and template declaration files are stored, and also to maintain status information such as which files have been turned into HTML pages, and which pages and handcrafted files have been uploaded to the production Web site.

Content and the content database

Content is the key to the Web site. Content can either be directly coded on to a handcrafted HTML page, or it can come from an external source, such as a database. Exclamation focuses on external content and how this is merged with a template to produce Web pages.

Content sources are any application or document that can produce a "tab-delimited file." Types of content include a variety of text types, numbers and dates. You have a great deal of control over text, with support for plain text and HTML text, plus some in between. Likewise with numbers and dates: you can carry out minor calculations with accuracy. More types of content are coming as well, particularly multimedia.

As the Exclamation Designer's Guide explains, the Exclamation project includes a content folder and a site plan folder (covered in the following chapters). A typical content folder contains:

- The database (one or more Excel, FileMaker, Access or other application spreadsheets or tables). All that is required of these applications is that they can export the data into a tab-delimited file.
- The tab-delimited files (one per database table).
- A Content Declaration file listing all the content sources (the tab-delimited files). For each source, it provides a table which maps the columns (fields) in the tab-delimited file, and identifies the data type of each column. The tables and columns are named, so that the template files can refer to them.

Database

A content database is a fundamental element of a Exclamation generated Web site.



Types

Exclamation is agnostic to the database application. Any application able to export tab-delimited files can be used. Examples of content management applications supported are:

- spreadsheets: Apple's AppleWorks spreadsheet, Microsoft's Excel
- desktop databases: FileMaker's FileMaker Pro, Apple's AppleWorks database, Microsoft Access

Support is also envisioned for other content management tools as well, such as OmniGroup's OmniOutliner and full ODBC SQL database servers.

Each table in the Content Declaration file discussed in the next chapter refers to a specific tab-delimited file containing the content material. These files are typically generated from a database, but can also be hand-crafted if it makes sense to do so.

A tab-delimited file looks just like a spreadsheet with the columns separated by tabs. Each row represents a record, and the columns are the fields, or attributes, that are collected for each record.

```
Fred   Flintstone   Bedrock       USA
Robin  Hood           Sherwood ForestEngland
        Shrek
Count  Dracula                Transylvania
```

This example illustrates a number of points about tab-delimited files:

- Cells can contain any punctuation except a tab. "Sherwood Forest" contains a space. Whole sentences or paragraphs are allowed, with any character except for a tab.
- A cell can be empty.
- Count is a title, but we elected to place the title in the First Name column. We could have created a new column for title, but we would then have had to give all the other rows (records) titles also, or add the tab so that the title column was skipped correctly.

Formats

The only format acceptable for database tables is a tab-delimited file. Multiple files are permitted, and there is no limit to the number of files.

Using more than one database

A database is a collection of related files. Exclamation permits content to come from more than one database, and even to be able to link records together from different databases. This allows databases to be controlled by different organizations, yet they can all contribute to the Web site.

As an example, an art gallery may have a database of artists and their works. The gallery manager keeps this information up-to-date. Meanwhile an assistant is responsible for the corporate art program — putting works on temporary display in the foyers of local corporate offices. The assistant uses the same database, but in addition maintains a separate and independent personal spreadsheet-based database containing all the data related to the corporate art program, including contact names and addresses, loan dates, and so on.

Content Declaration file

The Content Declaration file is the road map for content. No matter where the content may be stored on your computer network, inside the project or elsewhere, the Content Declaration document tells Exclamation where to find it and how to access each part of it.

The content declaration lists all of the content sources used by Exclamation. It declares the tables that are available for use in template queries, and where the data comes from that is in those tables. Here's a sample content declaration:

```
<?xml version="1.0" ?>
<!DOCTYPE content SYSTEM "jar:/Content.dtd">
<content>
  <tabdatabase>

    <folder base="content">db/movies</folder>

    <table name="bug" file="bugs.txt">
      <column name="id" type="plaintext" />
      <column name="title" type="paragraph" />
    </table>

  </tabdatabase>

</content>
```

This declares one content database, a folder with one flat file in it. The flat file contains tab-delimited column values. Each column has a



type mapping, which directs Exclamation to convert the source column type to a Exclamation type. You can declare more flat files in this folder using more `<table>` elements. You can also declare additional folders using more `<tabdatabase>` elements.

Paragraph text is parsed into paragraphs. That means multi-line plain text paragraphs are joined into a single line, and blank lines become a paragraph break. Control-K (character code 11) within a field is converted to a line feed when the data is pulled in.

Columns of a table are named and typed for Exclamation use.

The external name of a column does not have to match the Exclamation name. The benefit of this is to allow access to names which may not be legal in Exclamation. Its type can also be converted into something more useful for page generation. In many cases this reduces the often implementation-specific rich set of types to a simpler and smaller set of types. An example of this is converting all of the variously sized integer types into a single "integer" type.

In some other cases this may add additional meaning to the external type. An example of this is converting any character type as one of "plaintext", "paragraph" or "html".

Only the tables used by the templates need to be declared. The source may have more tables than are needed. Likewise, only the columns needed for page generation need to be declared.



XML Syntax

The first two lines of the content declaration must be:

```
<?xml version="1.0" ?>
<!DOCTYPE content SYSTEM "jar:/Content.dtd">
```

After these comes the `<content>` element:

```
<content>
...
</content>
```

<content>

There must be exactly one <content> element. Inside the content element are database elements. Currently only one kind of database is supported, the <tabdatabase>.

<tabdatabase>

Exclamation currently supports one kind of database file format, a flat text file of tab delimited values. This tag declares access to a folder that contains tab delimited text files. (Exclamation will ignore any other files found here.)

The tab-file database is assumed to be in the Content folder of the project. If it is located elsewhere, (for example `db/movies` in the example on page 13) then the <folder> tag is needed to show this.

<folder>

The <folder> tag declares where the tab database folder is. If omitted, it is assumed to be the Content folder set up in the application user interface. Most clients will want it done that way, to keep all Web project files together. The <folder> tag has one attribute and text content:

```
<folder base="basefolder">folderpath</folder>
```

The `basefolder` value can be one of three keywords.

absolute: The `folderpath` is an absolute path name. On Mac OS X, the name begins with a slash. On Windows the name begins with a drive letter, a colon and a backslash. Examples:

Mac OS X

```
<folder base='absolute'>/Inventory/exported</folder>
```

```
<folder base="absolute">
/Volumes/SystemB/Inventory/exported
</folder>
```

Windows

```
<folder base="absolute">Z:\INVENTORY\EXPORTS</folder>
```

content: The folderpath is relative to the Content folder configured in the application user interface. If folderpath is empty, then the Content folder itself holds the tab delimited files. Examples:

```
<folder base="content"></folder>
```

Gives the path (projectFolder)/Content.

```
<folder base="content">Inventory</folder>
```

Gives the path (projectFolder)/Content/Inventory.

```
<folder base="content">Pictures</folder>
```

Gives the path (projectFolder)/Content/Pictures.

user: The folderpath is relative to the user's home folder. If folderpath is empty, then the user's home folder itself holds the tab delimited files. Examples:

Mac OS X

```
<folder base="user"></folder>
```

```
<folder base="user">Inventory</folder>
```

Gives the path /Users/ (username) /Inventory.

Windows NT

```
<folder base="user">Inventory</folder>
```

Gives the path C:\WINNT\PROFILES\ (username) \Inventory.

Windows 2000/Windows XP

```
<folder base="user">Inventory</folder>
```

Gives the path C:\Documents and Settings\user\Inventory.

If you omit the base attribute, the default is "content". If you omit the <folder> element altogether, the default is the project's Content folder.

<table>

The <table> tag has three attributes to define the Exclamation name, content source, and external name.

```
<table name="Exclamationname" file="externalname">
```

The *Exclamationname* is the name you will use in your template declarations (see the <query> tag's "table" attribute on page 29). This name follows the same syntax as symbol names. A good rule of thumb is to use a plural word form for the name. So, if the table contains information about products, the name would be "products", not "product". Later, when you define a query for this table, you might name the rowlist of the query in the singular, since that name is used to access only one row at a time.

The *externalname* is the name of the table as defined by the source. For a tab-delimited file database, the external name is the tab-delimited file name. On Mac OS X and Windows, the file name is not usually case-sensitive. On Unix the name usually is case-sensitive.

<column>

Within each table element, you define the columns that you want to use. You give each column its Exclamation name and data type, and specify where it comes from in the external table. The pattern for this is:

```
<table name="movies" file="movies.txt" >
  <column name="Exclamationname" from="extname" type="typedeclaration"/>
  ...
  <column name="Exclamationname" from="extname" type="typedeclaration"/>
```

```
</table>
```

The *Exclamationname* of a column is the name you will use in your templates. This name follows the same syntax as symbol names.

The *extname* is the name defined by the external table. For tab-delimited files, the name is actually a column number. Columns are numbered left to right starting from 1. Since it is error-prone to have to type the numbers yourself, you can omit this attribute. The column number will implicitly be one more than the last column number, or 1 if it is the first column.

The *typeDeclaration* is a declaration of what Exclamation type this column has. Whatever type the external column has, the data in the column will be converted, within reason, to the Exclamation type. *Safetext* is the default type for text columns, which is assumed if *type* is omitted from the `<column>` tag.

The various data type values for the `<column>`'s *type* attribute are discussed in "Data types for `<column>` data" on page 19 which has a full specification of the syntax and meaning of the *type* attribute value.

<subtable>

There is sometimes a need to store a table inside another table.

Syntax

In the content declaration document, the `<table>` element can contain `<column>` elements and/or `<subtable>` elements.

```
<subtable name="name" separator="character" from="from">
  <column .../>
</subtable>
```

The column list is both a column and a kind of table. As a table, it has exactly one column. You declare the inner column's name and type using a single `<column>` tag. A `<subtable>` element takes the place of a `<column>`

element: it is can be thought of as a special column. Rows in the subtable are separated by the given character.

The default row separator is a space. The "from" attribute defaults as it does for the <column> tag. The name attribute is required.

FileMaker repeating fields can be used in a <subtable> column. The repetitions are separated by character code 1D (hex). XML disallows this code, so use %1D or %001D as the separator attribute. The latter allows for any Unicode/UCS-2 characters to be used. For example:

```
<subtable name="name" separator="%001D" from="from">
  <column name="number" type="integer" />
</subtable>
```

Data types for <column> data

The <column>'s `type` attribute accepts a variety of data types. Safetext is the default type for text columns, which is assumed if `type` is omitted from the <column> tag.

Some types support input formatting specifications. A format specifier begins with a colon.

Square brackets indicate the enclosed text is optional. Italics are used to indicate a complex rule described in the text.

type=date[:*format*]

A date and time. That is, both a date and time of day can be stored together in this column. Since dates and times are expressed in many ways, by using the format declaration you tell Exclamation how to interpret what it finds in the content source. The default is `date:short short`

The format declaration can either be one or two keywords (data keyword only, or date and time keyword), or it can be a custom format. The keywords are convenient shortcuts to use instead of making a custom format. Keywords and custom formats cannot be mixed. You may use a keyword if the column

contains a date or a date and time. A custom format must be used if the column contains only a time, or if the time precedes the date.

The content source must rigidly meet the spacing and punctuation requirements of the *format* declaration. Case is always insensitive for text, like AM/PM.

Some example column declarations using the date type are:

```
<column name="tourBegins" type="date:short"/>
<column name="publishedOn" type="date:long"/>
<column name="when" type="date:short short"/>
<column name="startingTime" type="date:h':'m' 'a"/>
```

In the first two examples, the content source column contains only a date. The time will be assumed to be midnight at the start of the day (12:00 a.m.). In the third example the content source column contains both a date and a time. In the fourth example, the content source column contains only a time, in 12-hour notation.

The two tables below list the keywords you can use for date and time.

DateKeyword	Meaning in US locale	Examples of allowed database values, US locale
Short	A completely numeric date. Year may be abbreviated.	2/10/56 2/10/1956 2/7/56
Medium	Month name may be spelled out or abbreviated.	Feb 10, 1956 February 10, 1956
Long	Exactly the same as Medium.	Feb 10, 1956 February 10, 1956
Full	The weekday, month, day and year. The month and weekday names may be abbreviated. The weekday is ignored if it disagrees with the month, day and year.	Friday, February 10, 1956 Fri, February 10, 1956 Fri, Feb 10, 1956

TimeKeyword	Meaning in US locale	Examples, US locale
Short	The time in hours and minutes. In the US locale, an AM or PM indicator is required.	<code>date 9:10 am</code> <code>date 7:37 PM</code>
Medium	The time in hours, minutes and seconds. In the US locale an AM or PM indicator is required.	<code>date 7:37:01 am</code> <code>date 7:37:01 PM</code>
Long	The time in hours, minutes and seconds, with a time zone. In the US locale an AM or PM indicator is required.	<code>date 9:10:32 AM EST</code> <code>date 7:37:00 AM PST</code> <code>date 7:37:00 AM GMT</code>
Full	Exactly the same as <i>Long</i> .	<code>date 9:10:32 AM EST</code> <code>date 7:37:00 AM PST</code> <code>date 7:37:00 AM GMT</code>

Custom formats use an advanced syntax as documented in Sun's Java API¹. Since this is a complicated document, the table below lists the most useful custom formats.

Keyword	Meaning in US locale	Examples, US locale
<code>d'.'M'.'y</code>	European style. Year may be abbreviated.	<code>10.2.56</code> <code>10.02.1956</code>
<code>dd 'MMM' 'yy</code>	European style with month names. Names may be abbreviated.	<code>10 Feb 56</code> <code>07 february 1956</code> <code>7 FEB 1956</code>
<code>M'-'d'-'y</code>	Numeric date separated by dashes. Year may be abbreviated.	<code>2-10-56</code> <code>2-10-1956</code> <code>2-7-56</code> <code>02-07-56</code>
<code>ddMMyyyy</code>	All-digit format with leading zeroes.	<code>10021956</code> <code>07021956</code>
<code>ddMMyy</code>	All-digit format with leading zeroes. Year may be abbreviated.	<code>100256</code> <code>10021956</code>
<code>H' : 'm</code>	Time only. 24 hour clock. Date is fixed at January 1, 1970.	<code>9:56</code> <code>13:04</code>
<code>H' : 'm' : 's</code>	Time only. 24 hour clock with seconds.	<code>9:56:23</code>

1. <http://java.sun.com/j2se/1.3/docs/api/java/text/SimpleDateFormat.html>

Keyword	Meaning in US locale	Examples, US locale
H': 'm': 's': 'S'	Time only. 24 hour clock with seconds and milliseconds.	9:56:23.001
M'- 'd'- 'y' 'H': 'm'	Date and time.	02-07-56 09:56
h': 'm' 'a'	Time only, with meridian.	9:56 am 10:04 pm
h': 'm' 'a' 'z'	Time only, with meridian and time zone	9:56 am gmt 10:04 pm EDT



type=decimal[:precision]

A fixed-point number. Fixed point numbers have some number of digits before the decimal point and some number after it. The benefit of fixed-point numbers is that arithmetic on them is as accurate as you wish, to as many decimal places as you need. This is also known as "precision." Numbers read from the external content source may define their own precision, such as including an explicit decimal point in the numeric text. For that kind of content you can omit :precision or specify it as ":*". Other content sources imply the precision, and do not have an actual decimal point stored in the external data. For this kind of content you specify ":integer" to show that the decimal point should be assumed to be before the last integer digits on the right.

The default precision is asterisk.

Exclamation type	External Value	Internal Value
decimal:*	1.2, 5, 3.14285	1.2, 5, 3.14285
decimal:2	1000, 0.025, 0.25	10.00, 0.03, 0.25
decimal:3	123456	123.456

type=html

Raw HTML text. Legal tags are inserted unchanged, but malformed tags are converted to safe text. This format is useful for handling HTML that has been cut from a source file and pasted directly into a database field.

type=integer

An integer, which is a whole number between -2 000 000 000 to +2 000 000 000.

type=paragraph

This type makes it easy for content contributors to enter paragraph breaks with no knowledge of HTML. Empty (or indented) lines become <P> tags.

Paragraph is identical to `safetext` except for the insertion of the <P> tags.

**type=plaintext**

Text. When substituted, it appears exactly as it was in the external source. It is possible to insert text that would break a generated HTML document, so use this with caution. Plaintext is a good choice if the content includes material that you need to remain untouched, for example JavaScript code.

type=safetext

Text. When substituted, it appears in the HTML page the way it was intended. That is, unsafe HTML characters are converted into safe representations. Less than (<), greater than (>) and ampersand (&) become `<`, `>` and `&`. When substituted into an attribute value, quote (") and apostrophe (') are similarly converted to `"` and `'`. This is the default type for text columns, which is assumed if `type` is omitted from the <column> tag.

When substituted into a URL value (such as "href" attributes), illegal URL characters are converted to `%xx` hex notation.

CONTENT DECLARATION FILE

Data types for <column> data

•
•
•
•

Template Declaration files

Templates are actually a pair of files, a Template Declaration file and its corresponding HTML template (or layout) file. Template Declarations are described in this chapter, and the HTML template files are discussed in Chapter 5.

See the Designer's Guide chapter on Creating a Web Site for descriptions of how the templates relate one to the other.

A Template Declaration declares three things:

- name and location of the HTML template file (the input file)
- name and location of the generated output file or files
- symbols that are used in the Template commands and substitution expressions

A sample declaration:

```
<?xml version="1.0"?>
<!DOCTYPE template SYSTEM "jar:/TemplateDeclarations.dtd">
<template>
  <input file="index.html" />
  <output file="index.html"/>

  <query table="artists">
    <rowlist name="artist"/>
  </query>
  <expression name="onStaff">artist.staff = "X"</>

</template>
```



Any *expression* elements and query declarations can appear in any order relative to each other. The physical ordering does not matter even if a query has a keep or omit expression which uses a symbol (query or expression) defined elsewhere in the document.

Example 1:

```
<query table="artists" sortBy="Order">
  <omit if="various"/>
  <rowlist name="all_artists"/>
</query>
<expression name="various" value="all_artists.id eq 'various'"/>
```

Example 2:

```
<expression name="various" value="all_artists.id eq 'various'"/>
<query table="artists" sortBy="Order">
  <omit if="various"/>
  <rowlist name="all_artists"/>
</query>
```

A `<keep>` or `<omit>` expression can reference another list. This is analogous to an SQL JOIN. A second list is created over and over for each row in the first list. For instance:

```
<query table="invoice" sortBy="number">
  <keep if="invoice.customer_id eq customer.id"/>
  <rowlist name="invoice"/>
</query>
```

Queries can appear in any order in a Template Declaration document. Even if one query has a keep or omit expression that uses columns from a second query, the physical ordering in the document is not important.

In order to compute the `<keep>` expression, which uses `customer.id`, the customer list must be "open" in an "active loop". Lists are opened using a LOOP command or its equivalent.

XML Syntax

Text shown in italics must be replaced by an actual value. A vertical bar ("|") indicates a mutually exclusive choice—choose one and only one of the alternatives.

<template>

Required. Indicates that a Template is being declared.



<input file="filepath" />

Optional. Identifies the HTML template file. *filepath* specifies the location and name of the HTML template file relative to the Template Declaration file. For clarity, it is generally best to use the same name for the HTML template and Template Declaration file. If the tag is not specified, the template document is assumed to be an HTML file (.html) with the same file name as the Template Declaration file.

<output file="filepath" | "filepathexpression" />

Optional. Describes the output (generated) document(s). If this tag is not specified, the output file name and extension are assumed to be identical to the input file name and extension (and only one file will be generated).

filepathexpression specifies the location and name of each file generated from the Template. The path is specified relative to the Template Declaration file. The actual output folder is at the corresponding location in the Preview folder.

If an explicit file path and name are specified, a single file is generated.

Example:

```
<output file="index.html"/>
```

If the Template Declaration file was, for example, `index.extd` then the generated file will be `index.html`

The path and/or name may contain one or more Exclamation substitution expressions that refer to columns in a single list. In this case, one file is

generated for each row in the list. (For more details on substitution, see “Substitution formatting” on page 68.)

Restriction: The list cannot depend on another list unless you use the `loop` attribute.

Example:

```
<output file="[page.id].html">
```



LOOP attribute in <output> tag.

If the output name expression has multiple lists, use the `LOOP` attribute to specify exactly which lists need to be looped, and in what order, to produce the generated output file names. The attribute value is a list of list names, separated by commas. The `LOOP` attribute is optional if the output tag includes only one list name.

Here is an example which produces one page per department and priority (so the most important products can be given prominence):

```
<output file="[dept.department]/[pri.priority].html" loop="dept, pri"/>
<query table="products" sortby="department,priority,name">
  <segmentlist name="dept" byequal="department">
    <segmentlist name="pri" byequal="priority">
      <rowlist name="product"/>
    </segmentlist>
  </segmentlist>
</query>
```

This is like defining two loops outside of the template (if that could be done):

```
<TLB LOOP="dept">
<TLB LOOP="pri">
  ...render one file...
</TLB>
</TLB>
```



If the output loop attribute is omitted, it is by default the name of the one list used in the file name expression.

The value may contain list names which are not directly used by the output file name expression. For example:

```
<output file="[dept.department]"/[[product.name]].html" loop="dept, pri, product"/>
```

This output file name expression produces one page per product. Since the product row list depends on the `pri` segment list, all three had to be listed.

<expression name=" *exprname*"> *expression* </expression>

Declare a new symbol (*exprname*) that can be used within the Template Declaration or the associated template file. `<expression>` elements can be placed before or after `<query>` elements.

expression specifies the value of the expression symbol. Since this expression is evaluated each time it is referenced, the value of the expression may change based on the context in which it is used (for example within a Template loop).

An expression can reference any symbol, including other expressions and any list. An expression can reference symbols before those symbols are declared. If an expression references a list, the expression symbol can only be substituted inside a loop over that list. See "Exclamation syntax" on page 58 for a detailed specification of expression syntax and semantics.

<query table=" *tablename*" orderby=" *columnname asc | desc,...*" > </query>

In full:

```
<query table="tablename" orderby="columnname asc | desc,..." > <keep> <omit> <segmentlist>
<rowlist> </query>
```

"*tablename*" specifies the name of a database table declared in the currently active Content Declaration.

`asc | desc` will sort in ascending (the default) or descending order. Omit the qualifier to sort in ascending order.

"*columnname, . . .*" indicates that the database rows are sorted by the values in the specified columns. The first column is the primary sort key. If no sorting

is specified, the table rows will be processed in random order. This order may change from run to run. Consequently, between the `<query>` and `</query>` tag, explicit sorting is recommended.

More than one query may be declared.

<keep> *selection expression* **</keep>**

Specifies which rows are to be retained. The selection expression must include a reference to at least one column in the current table. The selection expression may also refer to columns in a list defined in another query.

Example:

```
<query table="works">
  <keep>works.ArtistId EQ artist.id</keep>
  <segmentlist name="groupof3" bycount="3">
    <rowlist name="work"/>
  </segmentlist>
</query>
```

To reference a column in the current table, use the notation "tablename.columnname". This is similar to the more usual "listname.columnname" notation, using instead the table name because there is no list name for the original rows of the table.

More than one `<keep>` may be specified, in which case, all `<keep>` results are combined using logical OR. (Each additional `<keep>` *includes* more rows.) If no `<keep>` is specified, all rows are selected. `<keep>` "elements" must immediately follow the `<query>` tag.

<omit> *selection expression* **</omit>**

Removes rows from the current query based on the selection expression. If more than one `<omit>` is specified, rows satisfying any of the selection criteria are removed (like `<keep>`, results are logical ORed. (Each additional `<omit>`

excludes more rows.) `<omit>` "elements" must immediately follow any `<keep>` tags.

Example:

```
<query table="artists">
  <omit if="artists.id EQ 'various'"/>
  <rowlist name="artist"/>
</query>
```

An `<omit>` selection expression is identical to a `<keep>` expression. The only difference is that `<omit>` rows are removed and `<keep>` rows are retained.

`<segmentlist name="listname" bycount="integer" | byequal="colname" >`

Segments the query results into groups of rows. "*listname*" specifies the name of the list of segments—the segments themselves, not their contents. This name will be referenced in loop or substitution expressions.

`bycount="integer"` specifies the criterion for creating the groups of rows as a fixed number of rows (*bycount*).

`byequal="columnname"` specifies the criterion for creating a group of rows that have the same value in a specified column. The `byequal` *columnname* should match the first column in the query's `orderby` list of columns.

Segments can be nested as deep as desired. If `byequal` grouping is specified, the segment *columnnames* should be in the same order as the query's `orderby` list of columns.

Example:

```
<query table="products" orderby="Category,Publisher,Name">
  <segmentlist name="category" byequal="Category">
    <segmentlist name="publisher" byequal="Publisher">
      <rowlist name="product"/>
    </segmentlist>
  </segmentlist>
```

```
</segmentlist>  
</query>
```

<rowlist name="*listname*" />

Specifies the name of the innermost list of database rows. If no segments are specified, the <rowlist> tag must appear between the <query> and </query> tags.

Examples:

Minimal Template Declaration, Doubly segmented query

Subtable queries

There is sometimes a need to store a table inside another table. This is done using the <subtable> declaration in the Content Declaration document (see “<subtable>” on page 18).

The syntax of a subtable query is:

```
<query list="(listname)" subtable="(subtablename)" ...>  
  <keep>  
  <omit>  
  <segmentlist>  
  <rowlist>  
</query>
```

The rest of the query is no different. It can be sorted, filtered, and segmented. If "sortby" is omitted, the subtable rows are ordered as they appear in the plaintable field.

Example:

```
<table name="person" file="person.txt">
```

```

<column name='id' type='integer' />
<column name='fullName' type='safetext' />
<subtable name='favColors'>
  <column name="colorName" type="safetext" />
</subtable>
<column name='age' type='integer' />
</table>

```

·
·
·
·

The table "person" has four columns, named id, fullName, favColors and age. The subtable "favColors" has one column, named colorName. The template declarations for using these tables might be:

```

<query table="person" sortby="fullName">
  <rowlist name="person" />
</query>

<query list="person" subtable="favColors">
  <rowlist name='favColor' />
</query>

```

Template expressions to use these are:

```

<TLB LOOP=person>
  [[person.fullName]]'s most favorite color is [[favColor[FIRST].colorName]].
  The whole list is:
  <TLB LOOP=favColor>
    [[favColor.colorName]]<BETWEEN>, <BEFORELAST> and
  <TLBEND>
<TLBEND>

```

Output:

Bob's most favorite color is blue.

The whole list is: blue, green and red.

Judi's most favorite color is purple.

The whole list is: purple, yellow, teal and pink.

Notice that the `favColor` list is not defined except inside the person loop. That is no different than if the `favColor` list were defined in its own table and joined back to the person using a `keep` element.

Template declaration examples

The following sample Template Declaration illustrates the tags and attributes that should appear in every Content Declaration:

Recommended minimal Content Declaration:

```
<?xml version="1.0"?>
<!DOCTYPE template SYSTEM "jar:/TemplateDeclarations.dtd">
<template>
  <input file="index.html" />
  <output file="index.html"/>
  <query table="artists">
    <rowlist name="artist"/>
  </query>
</template>
```

Note: the `<input>` and `<output>` elements are actually optional, but it is good practice to include them.

Doubly segmented query

The following Template Declaration illustrates two levels of segmenting, first by category, and then by publisher:

```
<?xml version="1.0"?>
<!DOCTYPE template SYSTEM "jar:/TemplateDeclarations.dtd">
```

```

<template>
  <input file="bookpage.html" type="html"/>
  <output file="bookpage.html"/>
  <query table="books" sortBy="Category,Publisher,Title">
    <segmentlist name="category" byequal="Category">
      <segmentlist name="publisher" byequal="Publisher">
        <rowlist name="title"/>
      </segmentlist>
    </segmentlist>
  </query>
</template>

```

Linking queries

The following Template Declaration is used to create a 3-column thumbnail page for each artist in the database:

```

1  <?xml version="1.0"?>
2  <!DOCTYPE declare SYSTEM "jar:/TemplateDeclarations.dtd">
3  <template>
4    <input file="index.html" type="html"/>
5    <output file="..[[artist.id]]/index.html"/>
6    <query table="artists">
7      <omit if="artists.id EQ 'various'"/>
8      <rowlist name="artist"/>
9    </query>
10   <query table="works">
11     <keep if="works.ArtistId EQ artist.id"/>
12     <segmentlist name="groupof3" bycount="3">
13       <rowlist name="work"/>
14     </segmentlist>
15   </query>
16 </template>

```

Notes:

The second query is reevaluated for each artist, keeping only the works for that artist.

The two queries can be in either order, although it is more logical for the primary query (artists, in this case) to appear first.

The following template declaration is used to create a site home page that draws information about three selected works from the Works table:

```

1  <?xml version="1.0"?>
2  <!DOCTYPE declare SYSTEM "jar:/TemplateDeclarations.dtd">
3  <template>
4    <input type="html" file="index.html">
5    <output file="[page.id].html">

6    <query table="homepage">
7      <rowlist name="page">
8    </query>

9    <query table="works">
10     <keep if="works.id EQ page.ExKeyWorkId"/>
11     <rowlist name="workEx" >
12   </query>

13   <query table="works">
14     <keep if="works.id EQ page.WorkId1"/>
15     <rowlist name="work1">
16   </query>

17   <query table="works">
18     <keep if="works.id EQ page.WorkId2"/>
19     <rowlist name="work2">
20   </query>
21 </template>

```

The homepage table has only one row. It is used for editing text on the home page.

These three queries are presumably used in the template file to display information about three special art works.

Notes:

The "homepage" table has only 1 row.

Each of the other lists selects a single row from the "works" table.

About circular definitions

A circular definition is an error in which some symbol is ultimately defined in terms of itself. This is similar to the sentences:

- A hog is a pig.
- A pig is a hog.

It is easy to create a circular definition. Here is an example with an erroneous reference to an inner list:

```
<query name="all_artists" table="artists" sortBy="Order">
  <omit if="artist.id EQ 'various'"/>
  <segmentlist name="group" byCount="3">
    <rowlist name="artist"/>
  </segmentlist>
</query>
```

The analysis of this is as follows:

- * The "all_artists" query has an `<omit>` expression that depends on the "artist" rowlist. This list can't be computed until "artist" has been computed.
- * The "group" `<segmentlist>` depends on the results of the "all_artists" query.
- * The "artist" `<rowlist>` depends on the group segment.

That leads to a circular definition because the group segments cannot be determined until the `<omit>` is evaluated, and the `<omit>` depends on the result of the group segmenting. A diagnostic message is displayed.

This example can be corrected simply by having the `<omit>` expression refer to the query instead of the rowlist:

```
<query name="all_artists" table="artists" sortBy="Order">
  <omit if="all_artists.id EQ 'various'"/>
  <segmentlist name="group" byCount="3">
    <rowlist name="artist"/>
  </segmentlist>
</query>
```


HTML template files

HTML template files use HTML 4.01 with only a few additions. It is simple enough that most template files can be created without a programmer.

This simplicity provides for a cleaner-looking user interface (one that is not littered with programmer's code). Designers using WYSIWYG Web editors such as Dreamweaver and GoLive do not need to switch back and forth between WYSIWYG and source modes. Simple pages look just like the final page with substitution syntax in place of real text. Character entities (`&char;`) and URL hex escapes (`%xx`) are decoded. This is important for characters like quote and square brackets. Also, a cleaner user interface translates into cleaner previews in browsers.

Exclamation figures out where to place loops for common structures like sections, tables and lists. The ability to take these implicit commands, informally known as DWIM (Do What I Mean), is a unique Exclamation feature that finally makes it possible for a designer to write significant templates without the need for a programmer.

Substitution

A substitution is an expression enclosed in double square brackets `[[]]`. The expression must produce a number, text, or boolean expression. See Chapter 6 "Expressions" on page 59, for a complete description of expression syntax.

There are two forms for substitution:

Syntax of Simple Substitution:

```
[[substitution-expression]]
```

Examples of Simple Substitution:

```
The [[product.name]] was first introduced on  
[[product.date_of_introduction]].
```



Output:

The Triple-Frosted Donut was first introduced in 1992.

Syntax of Formatted Substitution:

```
[[substitution-expression:formatting-expression]]
```

The left expression is formatted according to the instructions of the right expression. Whether the formatting expression is present or not, the substitution expression result is formatted according to the rules of Substitution Formatting (see “Substitution formatting” on page 68).

Example of Formatted Substitution:

```
Price: [[product.price:"$0.00"]]
```

Output if product.price is not empty:

```
Price: $7.95
```

Output if product.price is empty:

```
Price:
```

When substituted, the value is converted to styled text, unless the value comes from a plaintext column. The resulting text is substituted into the generated output document according to the rules for styled text. Raw text is substituted as-is. If the expression's value is empty, nothing is substituted.

Note that the `vanish` command attribute can be used to eliminate the “empty substitution” so that the Price: text does not appear unless there is a value. See “VANISH command attribute” on page 45 for more details.

Predefined Symbols

There are two symbols that are predefined, `today` and `output`. The first is the current date and time, the second the output file name. These can be overridden with your own declarations, because they are not reserved names.

Note that templates which insert `[[today]]` will cause Exclamation to think your pages are all changed, simply because you re-rendered them.



Output symbol

The output symbol provides the relative path name of the generated file. It shows the generated path relative to the Preview folder. Typical uses include inserting the output file path into a constructed URL, perhaps to tell a CGI where to return, or to self-label a page.

Example 1:

```
<a href="/cgi-bin/buy.pl?item=[[prod.sku]]&returnto=[[output]]">Buy</a>
```

Output:

```
<a href="/cgi-bin/buy.pl?item=2303&returnto=shoes/203.html">Buy</a>
```

Example 2:

```
<!--[[output]]-->
```

Output:

```
<!--index.html-->
```

Exclamation HTML extensions

Exclamation commands are an attribute of everyday tags. Only one command can be used in a tag.

Exclamation “commands” in HTML tags

Exclamation “commands” appear as an extra attribute in standard HTML tags. A command attribute may be used inside almost any tag.

Exclamation TLB tag

Sometimes the content you want wrapped in a Exclamation command are not conveniently enclosed inside an HTML element. The special HTML tag <TLB> is provided for this purpose. It does nothing but hold a command attribute. The tag is not transferred to the output file.

Syntax:

```
<tlb command-attribute> content </tlb> | <tlbend>
```

Wherever possible, **command-attributes** should be placed inside ordinary HTML tags. There are times when this is not possible, for example when a loop encloses a number of different tags, or when the substitution you want to achieve should only affect a specific section in a text string. The <tlb> tag provides a way to enclose this material.

There are two ways to close the tag, </tlb> and <tlbend>. By using the </tlb> tag, some WYSIWYG editors (notably GoLive) are able to ignore the tag. However, many editors (including both Dreamweaver and GoLive) want to split the tag by inserting another </tlb> before a return, and another <tlb> tag after it.

The alternative is to use the <tlbend> tag, which is treated as an independent tag by the editors and therefore does not get split. As noted above, these unmatched tags appear as unrecognized tags in the WYSIWYG window of some editors. Consider using this second alternative for initial drafts of the page, and then as the structure firms up and you find yourself using the WYSIWYG window more than the source code window, make a global change and switch to the </tlb> tag.

In general and in principle we recommend the consistent use of <tlbend>, and this is what we use in our examples.

Example

```
<tlb loop="name"> [[substitute-expression]] <br> <tlbend>
```

Restriction

Exclamation ignores all embedded HTML comments. The good news is that this allows you to continue to exploit the power of Microsoft ASP and other third-party applications that are embedded in the comment tags. The bad news is that ignoring the content means that Exclamation tags cannot be embedded in these commands.

Conditional commands

IF Command Attribute

For the `if` command, if the expression is true, the whole element is left in the output document, minus the `if` attribute.

Syntax for IF:

```
<tag IF="expression"> content </tag>
```

For the `ifnot` command, if the expression is false, the whole element is left in the output document, minus the `ifnot` attribute.

Syntax for IFNOT:

```
<tag IFNOT="expression"> content </tag>
```

Example 1

Example of odd/even row coloring:

```
<table>
<tr bgcolor="lightblue" IF="PositionOf(myList) % 2 EQ 0">
<td>[[book.title]]</td><td>[[book.isbn]]</td>
</tr>
<tr bgcolor="white" IFNOT="PositionOf(myList) % 2 EQ 0">
<td>[[book.title]]</td>
<td>[[book.isbn]]</td>
</tr>
</table>
```

In this example, the row colors will alternate between white and yellow. The exact same `IF`-expression is used for each `<tr>`, changing only `IF` to `IFNOT`. The percent symbol (%) provides the remainder of division by 2. That can only

be 0 or 1. If the remainder is 0, then this is an even-numbered row. If it is 1, this is an odd-numbered row.

Example output:

```
<table>
<tr bgcolor="#fffc9e"><td>XML: The Annotated Specification</td>
<td>0-13-082676-6</td></tr>
<tr bgcolor="white"><td>Web Content Management</td>
<td>0-201-65782-1</td>
</tr>
<tr bgcolor="#fffc9e">
<td>Cocoa programming for Mac OS X</td><td>0-201-72683-1</td>
</tr>
</table>
```

Example 2

Example of hiding an empty table:

```
<TLB IF="NumberOfRows(book) GT 0">
<p>Reading list for the [[semester.season]] '[[semester.year]] semester:</p>
<table>
<tr>
<td>[[book.title]]</td><td>[[book.isbn]]</td>
</tr>
</table>
<TLBEND>
```

In cases where the most important substitutions will not occur (for example `book.title` in the example above) use `if` to test the important condition. `Vanish` would not work here since the first two substitutions from the `semester` would cause the empty table to be shown.



VANISH command attribute

If all substitutions in the element (or content or tag) are empty, then the element (or content or tag) is eliminated. All substitutions must be empty, otherwise the whole element is substituted into the output document as usual.

Syntax for VANISH:

```
<tag VANISH=ELEMENT> content </tag>
```

```
<tag VANISH=CONTENT> content </tag>
```

```
<tag VANISH=TAG> content </tag>
```

"Tag" refers to the start tag and the end tag; the substitutions examined are those in the tag's attribute values. If they are all empty, then the start tag and end tag are removed, substituting just the element content.

"Content" refers to the markup and text between the start tag and the end tag; the substitutions examined are the ones in the content. If they are all empty, then the content is removed, leaving the start and (if present) end tag.

"Element" refers to the start tag, the content, and the end tag; the substitutions examined are the ones in the entire element. If they are all empty, then the entire element is removed, substituting nothing.

Example 1 of VANISH:

```
<A HREF="[[company.url]]" VANISH=TAG>[[company.name]]</A>
```

Output if company.url NE ""

```
<A HREF="http://www.apple.com/">Apple Computer, Inc.</A>
```

Output if company.url EQ ""

```
Apple Computer, Inc.
```

Example 2 of VANISH:

```
<TR><TD VANISH=CONTENT>Price: [[item.price:"$0.00"]]</TD></TR>
```

Output if item.price NE ""

```
<TR><TD>Price: $23.50</TD></TR>
```

Output if item.price EQ ""

```
<TR><TD></TD></TR>
```

Example 3 of VANISH:

```
<TLB VANISH=ELEMENT>Price: [[item.price: "$0.00"]]</TLB>
```

Output if item.price NE ""

```
Price: $23.50
```

Output if item.price EQ ""

```
(nothing)
```

Looping commands

Loop command attribute

For the LOOP command, the whole element is substituted once per list item. Within the element, the list's CURRENT, FIRST, LIST, NEXT and PREVIOUS row subscripts are defined.

Syntax for LOOP:

```
<tag LOOP=list-symbol> content </tag>
```

Example 1 of LOOP:

```
<TABLE><TR LOOP=product>
  <TD>[[product.name]]</TD>
  <TD>[[product.price]]</TD>
</TR></TABLE>
```

Output if list has rows

```
<TABLE><TR>
  <TD>Gadget Fremulator</TD>
```

```
<TD>165</TD>
</TR><TR>
  <TD>Widget XJ23</TD>
  <TD>25</TD>
</TR></TABLE>
```

Output if list is empty

```
<TABLE></TABLE>
```

Example 2 of LOOP:

```
These are on sale: <UL>
<TLB LOOP=sale><LI> [[sale.productName]]
</TLB></UL>Sale ends on Sunday.
```

Output if list has rows

```
These are on sale:<UL>
<LI> Wingtip shoes
<LI> Dance shoes
<LI> Suede shoes
</UL>Sale ends on Sunday.
```

Output if list is empty

```
These are on sale:<UL>
</UL>Sale ends on Sunday.
```

BETWEEN and BEFORELAST attributes

Used with the LOOP command attribute, the value of the *between* attribute specifies text which should be inserted between each iteration of the loop.

If there is also a `beforelast` attribute, then its text is inserted between the second to last and last iteration. This is done instead of the `between` attribute value for this last iteration only.

Syntax for BETWEEN and BEFORELAST attributes:

```
<tag LOOP=list-symbol BETWEEN="text1" BEFORELAST="text2"> content </tag>
```

This “simple” form works well for plain text, or just a couple of characters. But as examples below will illustrate, as soon as the text starts to get complex (especially if HTML tags are involved) the second form is recommended.

Syntax for BETWEEN and BEFORELAST tags:

```
<tag LOOP=list-symbol> content <BETWEEN> text1 <BEFORELAST> text2 </tag>
```

The `between` and `beforelast` attributes are optional, as are their tag forms. `Between` can be used without `beforelast`. Without either, the content is repeated for each row of the list:

```
content content ... content
```

With `between` and `beforelast`, the content and text parts are repeated this way:

```
content text1 content text1 ... content text2 content
```

The text string can be anything, including HTML, or perhaps an image. Note however that it is a literal text string and so the code would need to be spelled out (see example of special text values below).

```
<tag LOOP=list-symbol BETWEEN=text BEFORELAST=text> content </tag>
```

Example of BETWEEN and BEFORELAST:

We ship to:

```
<TLB LOOP=place BETWEEN=", " BEFORELAST=", and ">[[place.state]]</TLB>.
```

Output if list has rows

We ship to Maryland, Ohio, and Pennsylvania.

Output if list is empty

We ship to .

The text values are processed as plaintext. After normal HTML processing, the resulting text is inserted as-is. (Recall that normal processing means to process character entities and unescape hex characters encoded using %xx notation.)

Example of special text values:

```
We ship to<br><TLB LOOP=place BETWEEN="&lt;BR&gt;">[[place.state]]
</TLB>.
```

Output if list has rows

We ship to
Maryland
Ohio
Pennsylvania.

Output if list is empty

We ship to
.

We strongly recommend using the second syntax form rather than attempting to keep the special text values legal.

Implicit looping

In order to write simple templates entirely in WYSIWYG mode, Exclamation can imply loops around "expected" HTML elements. There are many situations where Exclamation can correctly guess where loops are needed, for example over rows in a table, or over a bulleted list.

Implicit looping makes it easy to create simple displays of lists. The most likely uses of loops, around `<tr>` tags and `` tags, can be simplified to the point of

not needing to literally write a `loop` command attribute. The benefits are faster template design, quicker learning, and improved opportunities for WYSIWYG editing.

Here are some examples of loops, with explicit loops and with implicit loops.

Explicit:

```
<table>
  <tr LOOP="movie">
    <td>[[movie.title]]</td>
  </tr>
</table>
```

Implicit:

```
<table>
  <tr>
    <td>[[movie.title]]</td>
  </tr>
</table>
```

Explicit:

```
<ol>
  <li LOOP="movie">[[movie.title]]
</ol>
```

Implicit:

```
<ol>
  <li>[[movie.title]]
</ol>
```



As you can see, the mere fact that the `movie` substitution is not enclosed in a `loop` tells Exclamation to try and place a `loop` attribute where it is expected. The rules are laid out below.

Rule 1: Why an implied loop is needed

If a substitution requires a list symbol, and the substitution is not in the body of a loop, then an implied loop is created.

Rule 2: Start the loop at the nearest loopable tag

When an implied loop is created, it is associated with the nearest preceding tag from Table 1. The search for a free tag cannot extend outside an explicitly defined loop (a tag with a "loop" attribute).

Loopable Tag	Scope
LI	The scope is just the LI element (which ends before next LI or /OL tag).
LI	The scope is just the LI element (which ends before next LI or /UL tag).
DT+DD	The scope is a series of zero or more DT elements followed by a series of zero or more DD elements. DT ends before the next DD, DT, or /DL. Same for DD, it ends before the next DD, DT, or /DL.
TR	The scope is the TR element and all TRs after it in the same table.
Hn	The scope is the Hn element plus all elements until the next Hm where $m \leq n$.

Rule 3: Implied loops can nest

If multiple lists are required, then the innermost dependent list is placed first, followed by the lists it depends on. Due to Rule 2, the innermost list is assigned to the nearest loopable tag, "A", and the next list is assigned to a loopable tag that contains that tag "A." This rule is applied only if a single expression needs multiple loops. Otherwise loops are created first and innermost, based on which list appears first in the template. It is best to use explicit loops if you have multiple lists.

Rule 3A: ...but they can't be inside explicit loops

It is not allowed for some of the lists to be explicitly in loops but others not. This is prohibited because it complicates the rules. There is just one exception:

the outermost loop may be the list used in the generator expression for the `<output file>` tag attribute. Technically this is still an implied loop which contains the entire `<html>` element.

Loopable tags cannot be assigned more than one `loop` attribute, whether explicit or implicit. Once a list is "assigned" to a particular loopable tag, then any further lists needing implicit loops will search for an earlier loopable tag.

Example 1

In which `grade` is a segmentlist and `student` is a rowlist in `grade`. **Bold** code is the implicit loop which is not needed:

Elementary school student roster, by grade levels:

```
<ul>
  <TLB LOOP=grade>
    <li>Grade [[grade.gradeNumber]]:
      <ul>
        <TLB LOOP=student>
          <li>[[student.fullName]] ([[student.teacher]])
        <TLBEND>
      </ul>
    <TLBEND>
  </ul>
```

Notice that the loop is not on the LI element itself, but rather just outside of it. Also notice the final `<tlbend>` appears after where the `` would be, which is implied by the `` closing tag.



Example 2

The same lists can be used in a definition list example:

Elementary school student roster, by grade levels:

```
<dl>
  <TLB LOOP=grade>
    <dt>Grade [[grade.gradeNumber]]:
    <dd>Students:
      <TLB LOOP=student>
        <ul>
          <li>[[student.fullName]] ([[student.teacher]])
        <TLBEND>
      </ul>
    <TLBEND>
  </dl>
```

Example 3

Here is an example using tables:

```
<table border=1>
  <TLB LOOP=grade>
    <tr><th colspan=2>Grade [[grade.gradeNumber]]</th></tr>
    <tr><th>Student</th><th>Teacher</th></tr>
  <TLB LOOP=grade>
    <tr><td>[[student.fullname]]</td><td>[[student.teacher]]</td></tr>
  <TLBEND>
</table>
```

Example 4

Finally, here is an example using headed sections. There is an additional list, coordinator, which only has one row. It is linked to the grade list by the gradeNumber column.

```

<html>
<head>
<title>Student Roster for Pleasantville Elementary School</title>
</head>
<body>
<h1>Student Roster for Pleasantville Elementary School</h1>
<p>Here is a list of the students by grade for the current year, [[today:"YYYY"]].

<TLB LOOP=grade>
<h2>Grade [[grade.gradeNumber]]</h2>
<p>The PTA coordinator for this grade is [[coordinator[FIRST].name]], phone
[[coordinator[FIRST].phone]].
<h3>Roster</h3>
<table border=1>
  <tr><th>Student</th><th>Teacher</th></tr>
  <TLB LOOP=student>
  <tr><td>[[student.fullname]]</td><td>[[student.teacher]]</td></tr>
  <TLBEND>
</table>
<TLBEND>
</body>
</html>

```

Important note: The use of coordinator [FIRST] is done because there is only ever one row in the list. If the subscript [FIRST] were omitted, an additional loop would be needed for the coordinator list. Implicit looping would not work in that case.

Student and grade are lists that come from that same query. By using a list from a different query, coordinator, implicit looping becomes trickier. If you find implicit looping is causing puzzling diagnostic messages, switch to explicit looping to see if that clears up the problem.

Common Syntax

This chapter documents all the rules of Exclamation syntax. But first, a word about case-sensitivity. Most names are not case-sensitive. XML tag names, XML attribute names, and XML attribute values are the only exception. XML element content (such as the expression between `<keep>` and `</keep>` is not case sensitive. Within HTML nothing is case-sensitive.

File naming conventions

There are no restrictions on file names, but adhering to the following conventions may save confusion:

- 1 Use the following extensions:
 - .excd for Content Declaration files
 - .extd for Template Declaration files.
- 2 Do not use double square brackets `[[]]` in a file name unless the file is a page layout template.

The discussion in the Exclamation Designer's Guide chapter on Creating a Simple Web Site illustrates these conventions and their usefulness.

HTML 4.01 review

It is assumed that you are annotating a valid HTML file. If you use Dreamweaver or GoLive the applications automatically validate the HTML code. If you are using a text editor, it is a good idea to perform a syntax check before adding Exclamation commands.





Before we look at how Exclamation interacts with HTML element structure, it is important to note the two "types" of HTML elements.

Begin and end tagged elements

Begin and end tags form an element if their tag names match. Elements may not partially overlap other elements, so the content in an element is entirely self-contained. Start and end tag names do not have to match the tag case. `<p>` matches `</P>`, for instance. If a start tag does not have a matching end tag inside the current element, then it is considered to be singular event (or if the end tag is required, it will cause an error).

In the following example, the element is the P start tag, the content and the P end tag. (A more compact way to say this is it is a P element.)

Example of simple element:

```
<p> content </p>
```

In the following example of nesting, there are two elements, P and B. The P element contains the B element.

Example of complex elements:

```
<p> content1  
  <b> content2 </b>  
content3 </p>
```

Begin-only tagged elements

An element ends when its enclosing element ends. It encompasses all events to the end of its containing element.

Example of begin-only tagged element:

```
<ul>  
  <li> content3  
  <li> content4  
  <li> content5  
</ul>
```

XML syntax

This section provides a very brief description of the XML syntax used in a Content Declaration or Template Declaration. XML knowledge is not essential in order to use Exclamation, but it provides a definite advantage. Many excellent books are commercially available for readers wishing more complete descriptions of XML. One we particularly recommend is:

- "*XML: The Annotated Specification*" by Bob DuCharme (Prentice Hall PTR, 1999, ISBN 0-13-082676-6)

Experienced XML programmers can skip this section.

XML is like HTML, except you must always use end tags, and you must quote all attribute values. Thus a document in XML format consists of pairs of opening and closing "tags" in the following format:

```
<tagname attributename="value" ...>element content</tagname>
```

An opening tag can include any number of attributes (including none). There must be white space (at least one space, tab, or line break) preceding each attribute name. Additional white space is purely cosmetic to make the XML easier to read. The end tag is not optional, the way it is in HTML. The three parts—*start tag*, *content*, and *end tag*—form an *'element'*.

Every attribute must have an associated value enclosed in quotation marks (apostrophes or double-quotes). Quotation marks are not optional, the way they are in HTML. White space in a value is significant.

As noted earlier, tag names and attribute names are case-sensitive.

XML allows the element content to be pure text, pure XML tags, or a mixture. Exclamation's use of XML restricts the element content to be pure text or pure tags, depending on the specific XML tag.

If there is no element content, the closing tag can be merged into the opening tag. The following two XML lines are equivalent:

```
<tagname attributename="value"></tagname>  
<tagname attributename="value"/>
```

XML documents used by Exclamation must begin with either of the following two lines:



```
<?xml version="1.0"?>
<!DOCTYPE content SYSTEM "jar:/Content.dtd">
```

or

```
<?xml version="1.0"?>
<!DOCTYPE template SYSTEM "jar:/Template.dtd">
```

The very first two characters of the file must be the "<?", with no blank lines or leading spaces.

Exclamation syntax

Symbols

Symbol references consist of names, keywords and perhaps some punctuation. Names must start with a letter and may internally contain letters, digits and underscore. Names and keywords are case-insensitive. Symbol reference forms are:

```
expressionname
listname.columnname
listname[FIRST].columnname (FIRST can also be LAST, PREVIOUS, CURRENT, NEXT or a number)
```

Constants

Constants can be integers or quoted text. The quoting character can be either quotation mark (") or apostrophe (').

Examples:

```
'YES'
"TRUE"
' '
""
```



1
-10.50

Expressions

Expressions exist in order to perform comparisons, arithmetic and string functions.

- Expressions can be used in three places in a Declaration file:

```
<expression name="symbolname"> scalar-value-expression </expression>
<keep> selection-expression </keep>
<omit> selection-expression </omit>
```

- As shown by *scalar-value-expression* in the first example above, an expression can define a scalar value, which can be used in other expressions or inserted as if it were a column from a database table.
- Expressions can also be used in the template *if* and *ifnot* functions, *if* and *ifnot* tag attributes, and in substitutions. We do not recommend using expressions in the HTML template file, because it is more useful to keep the template simple.

Expressions consist of:

- Operators. The comparison operators reduce all formatted text to plaintext formatting before doing the comparison.
- Functions. This is a list of the expression functions that operate on text (see "Functions" on page 61). Any use of plaintext in a function causes it to become styled text. More exactly, the function result is always styled.
- Expression symbols. The `<expression>` element defines a symbol for an expression. That symbol can be used in other expressions. This gives you a way to define a common expression or value in one place. It can be compared to a macro "named expression."

Note that the `<expression>` element defines a symbol not a value. The difference is that the value isn't known until the symbol is used. That way you can use a list name in the expression and have the expression symbol change value for each iteration in a loop.



Boolean expressions treat empty text (zero-length strings or all blanks) as false, numbers are false if they are zero. Error values occur if there is an error in the expression, or in the data fed into the expression. Error values are always false.

Operator syntax

Expressions are algebraic in nature. Valid expressions are composed of these parts:

- e1 AND e2
- e1 OR e2
- e1 EQ e2 (EQ can also be NE, GT, GE, LT or LE)
- e1 + e2
- e1 - e2
- e1 * e2
- e1 / e2 (using two integers will not produce fractional quotient)
- e1 % e2 (remainder of division)
- e (negate)
- (e) (parentheses change order of evaluation)
- NOT(e) (boolean complement)

e, e1, and e2 can be expressions, constants or symbol references.

Parentheses are strongly recommended to enforce the desired precedence.

Precedence levels

AND OR	Less binding/done last
EQ, NE, GT, GE, LT or LE	
+ -	
* / %	
- () NOT	More binding/done first

Example expressions:

```
product.price - (product.price * 0.10)
```

Subtract 10% from the list price.

```
product.price - product.price * 0.10
```

Subtract 10% from the list price, same as above. Precedence rules say that multiplications are done before subtraction. The use of parentheses makes this unquestionably right—so use parentheses.

```
positionof(transaction) % 2 EQ 1
```

This is true if the current row is odd, false if the current row is even.

```
positionof(movie) % 10 EQ 0
```

This is true if the current list position is row 10, 20, etc. All other positions are false.

```
movie.rating
```

This is the rating code (true) if the rating is not empty. It is empty (false) if there is no rating provided. See “IF Command Attribute” on page 43 for an explanation of how true and false are interpreted from expressions.

```
articlePage[NEXT].date
```

The first row's date of the next articlePage segment. This may be useful for producing forward navigation links.

```
articlePage[PREVIOUS].date
```

The first row's date of the next articlePage segment. This may be useful for producing backward navigation links.

Functions

Functions are 1-based (start counting at 1, not 0).

atfirst (*listname*)

The result is true if the current iteration of the loop for listname is at row 1.

Example:

```
<tr if="atfirst(person)"><td colspan=5>People:<td></tr>
```



atlast(*listname*)

The result is true if the current iteration of the loop for listname is at the last row.

Example:

```
<tr if="AtLast(transaction)"><td>Grand Total:</td><td>transaction.grandtotal</td></tr>
```



attribute(*text*)

Convert the text expression to plaintext, safe for insertion in an attribute value. Unsafe characters are converted to <;, >;, &;, "; and '; (apostrophe). The function name may be abbreviated as attr.

This function is useful when producing pages that contain ASP, JSP or JavaScript code.



decimal (*number*) and **decimal**(*number,precision*)

Convert the number expression to a fixed decimal value with no fractional digits. The result is rounded halfway up. The precision is the number of fractional digits the result will have. The default precision is zero. Use this function when performing financial calculations. It is necessary for controlling the number of fractional digits precisely.



Example:

decimal(# , p)	Result
58	58
58.2	58
58,1	58.0
58,2	58.00
58.2,0	58
58.2,1	58.2
58.2,2	58.20
58.55,0	59
58.55,1	58.6
58.55,2	58.55

hasrows(listname)

hasnorows (listname)

The result is true or false depending on whether the number of rows in the list is zero or not.

Example:

```
<p IF="HasNoRows(featured)>There are no featured products this week</p>
<table IF="HasRows(featured) ">
  <caption>Featured Products</caption>
  <tr LOOP="featured">...</tr>
</table>
```

if (condition, then, else)

If the condition expression is true, then the result is the value of the expression. Otherwise, the result is the value of the else expression. All expressions are evaluated, regardless of the outcome. See the "IF Command Attribute" on page 43 for the interpretation rules of the conditional expression.

isok (*expression*)

The result is true if the expression is correct. The expression is not correct if there is any error in it, including errors like division by zero or null values from a table.

Example:

```
<tlb if="IsOK(product.publicPrice)">${[product.publicPrice]}</tlb><tlb
ifnot="IsOK(product.publicPrice)">CALL</tlb>
```

mod (*dividend, divisor*)

The result is the modulus (unsigned remainder) of the result of dividing the divisor expression into the dividend expression.

Examples:

```
mod(5,2) is 1.
mod(-5,2) is 1.
```

Compare with this:

```
-5 % 2 is -1
```

numberofrows (*listname*)

The result is the number of rows in the list.

positionof (*listname*)

The result is the current iteration's row number in the list.

Example:

```
<tr class="rowcolor[[PositionOf(movie) % 5]]">
```

Example output:

```
<tr class="rowcolor0">...</tr>
<tr class="rowcolor1">...</tr>
<tr class="rowcolor2">...</tr>
<tr class="rowcolor3">...</tr>
<tr class="rowcolor4">...</tr>
```



```
<tr class="rowcolor0">...</tr>
<tr class="rowcolor1">...</tr>
<tr class="rowcolor2">...</tr>
<tr class="rowcolor3">...</tr>
<tr class="rowcolor4">...</tr>
<tr class="rowcolor0">...</tr>
<tr class="rowcolor1">...</tr>
...etc...
```

Here is an alternate coloring scheme:

Example 2

```
<tr class="rowcolor[[(PositionOf(movie) / 5) % 2]]">...</tr>
```

Example 2 output:

```
<tr class="rowcolor0">...</tr>
<tr class="rowcolor0">...</tr>
<tr class="rowcolor0">...</tr>
<tr class="rowcolor0">...</tr>
<tr class="rowcolor0">...</tr>
<tr class="rowcolor1">...</tr>
<tr class="rowcolor1">...</tr>
<tr class="rowcolor1">...</tr>
<tr class="rowcolor1">...</tr>
<tr class="rowcolor1">...</tr>
<tr class="rowcolor0">...</tr>
<tr class="rowcolor0">...</tr>
...etc...
```

Here rows are colored in groups of 5, using a total of 2 colors. The general formula is:

```
(PositionOf(listName) / rowsPerColor) % numberOfColors
```

(where you have defined CSS class names `rowcolor0` through `rowcolor4` to provide the background color for rows in groups of 5).



url(*text*)

Computes a copy of `text` with all the styling removed and the resulting text encoded for use in a URL. Letters and digits remain plain, but punctuation is converted to hex notation (`%xx`). Spaces are converted to hex notation (`%20`), not plus (+) because most browsers incorrectly leave plus signs as-is when processing `mailto` URLs.

Parameters:

`text`

The expression value must be a number or text. Numbers are converted to text using default formatting rules.

Returns:

`text`

Text with no styling and encoded for use in a URL.

This is useful when generating JavaScript or other pages containing special code or special tags. However it should never be used in URLs for HTML 4.01-standard tag attributes. These are implicitly processed by the URL function.

Example:

```
<a href="[[person.fullName]].html">
```

If `fullName` has a space in it, the output generated is properly escaped:

```
<a href="Pat%20Doe.html">
```



Decimal math

It is best to perform arithmetic outside of Exclamation, and store results in the database. However there is some simple arithmetic capability in Exclamation that can be useful.

You can compute numbers using fixed precision only. Floating point, or "real", numbers are not available. Integers have no digits after the decimal point, and decimal numbers do have these "fractional" digits. 123 is an integer, and 12.3 is a decimal number.

The number of fractional digits is the "precision" of the decimal number. Precision is controlled mainly by you, and does not change outside of your control.

10.000 is a number with three digits of fractional precision. The number of digits you write in a constant decides its precision.

A list and column reference, such as `product.price`, has whatever precision was declared for that column in the content declaration document.

You can change the precision of any constant, column or result using the decimal function. The expression `decimal(product.price, 2)` will make a copy of the price column and force the copied value to be precise to two fractional digits.

Whenever rounding must happen, the halfway value is rounded up to a larger value. Rounding 10.5 to zero fractional digits yields 11, and rounding -10.5 to zero fractional digits yields -11.

Addition and subtraction give a result with a precision that is only as large as the largest precision of the left or right value. $1.00 + 1.000$ gives 2.000. $1 + 1$ is 2.

Multiplication, the mod function and remainder(%) give a result whose precision is the sum of the left and right precisions. $4.00 * 2.000$ is 8.00000. $4 * 2$ is 8.

Division gives a result that is the same precision as the dividend (the left value). $4.00 / 2.000$ is 2.00, and $2.000 / 4.00$ is 0.500. $4 / 2$ is 2, and $2 / 4$ is 0.

A note of caution here: $1 / 2$ is 0, but `decimal(1,0)/2.00` is 1. When math is done strictly on integers, truncation happens. When decimal values are involved, rounding occurs. The latter expression is 0.5, but rounded to the precision of the dividend rounds it up to 1. If you have any doubts about what precision or result you'll get, you can use Exclamation to produce a little calculation for you.

There is no limit to the precision of a number.



Substitution formatting

Content is normally substituted into generated files with minimal formatting. This note describes the rules for creating a textual representation of the following types of binary data:

- numbers
- date and/or time

General syntax

```
[[substitution-expression : formatting-expression]]
```

The formatting expression on the right of the colon (:) is text, and normally a quoted text literal, such as "\$0.00" or '\$0.00' (see page 39). Even if there is no explicit formatting expression, default rules apply.

The instructions are written according to the data type of the substitution-expression. The two types supported are numbers and dates. Other types do not use explicit substitution formatting expressions.

Number formatting

The instructions for number formatting are taken directly from the Java class `com.text.DecimalFormat`¹.

Number patterns

A number pattern contains a positive and negative subpattern, a `DecimalFormat` for example, "`#, ##0.00; (#, ##0.00)`". Each subpattern has a prefix, numeric part, and suffix. The negative subpattern is optional; if absent, then the positive subpattern prefixed with the localized minus sign ('-' in most locales) is used as the negative subpattern. That is, "`0.00`" alone is equivalent to "`0.00; -0.00`". If there is an explicit negative subpattern, it serves only to specify the negative prefix and suffix; the number of digits, minimal digits, and other characteristics are all the same as the positive

1. <http://java.sun.com/j2se/1.3/docs/api/java/text/DecimalFormat.html>



pattern. That means that "#,##0.0#;(#)" produces precisely the same behavior as "#,##0.0#;(#,##0.0#)".

The prefixes, suffixes, and various symbols used for infinity, digits, thousands separators, decimal separators, and so on may be set to arbitrary values, and they will appear properly during formatting.

The grouping separator is commonly used for thousands, but in some countries it separates ten-thousands. The grouping size is a constant number of digits between the grouping characters, such as 3 for 100,000,000 or 4 for 1,0000,0000. If you supply a pattern with multiple grouping characters, the interval between the last one and the end of the integer is the one that is used. So "#,##,###,####" == "#####,####" == "##,####,####".

Illegal patterns, such as "#.#.#" or "#.###,###", will result in a warning diagnostic with the code "BADFMT".



Scientific notation

Numbers in scientific notation are expressed as the product of a mantissa and a power of ten, for example, 1234 can be expressed as 1.234×10^3 . The mantissa is often in the range $1.0 \leq x < 10.0$, but it need not be. In a pattern, the exponent character immediately followed by one or more digit characters indicates scientific notation. Example: "0.###E0" formats the number 1234 as "1.234E3".

- The number of digit characters after the exponent character gives the minimum exponent digit count. There is no maximum. Negative exponents are formatted using the localized minus sign, not the prefix and suffix from the pattern. This allows patterns such as "0.###E0 m/s".
- The minimum and maximum number of integer digits are interpreted together:
- If the maximum number of integer digits is greater than their minimum number and greater than 1, it forces the exponent to be a multiple of the maximum number of integer digits, and the minimum number of integer digits to be interpreted as 1. The most common use of this is to generate engineering notation, in which the exponent is a multiple of three, e.g.,

"##0.####E0". Using this pattern, the number 12345 formats to "12.345E3", and 123456 formats to "123.456E3".

- Otherwise, the minimum number of integer digits is achieved by adjusting the exponent. Example: 0.00123 formatted with "00.###E0" yields "12.3E-4".
- The number of significant digits in the mantissa is the sum of the minimum integer and maximum fraction digits, and is unaffected by the maximum integer digits. For example, 12345 formatted with "##0.##E0" is "12.3E3". To show all digits, set the significant digits count to zero. The number of significant digits does not affect parsing.
- Exponential patterns may not contain grouping separators.

Pattern syntax

```

pattern      := pos_pattern{';' neg_pattern}
pos_pattern  := {prefix}number{suffix}
neg_pattern  := {prefix}number{suffix}
number       := integer{'.' fraction}{exponent}
prefix       := '\u0000'..' \uFFFD' - special_characters
suffix       := '\u0000'..' \uFFFD' - special_characters
integer      := min_int | '#' | '#' integer | '#' ',' integer
min_int      := '0' | '0' min_int | '0' ',' min_int
fraction     := '0'* '#'*
exponent     := 'E' '0' '0'*

```

Notation:

```

X*           0 or more instances of X
{ X }       0 or 1 instances of X
X | Y       either X or Y
X..Y       any character from X up to Y, inclusive
S - T      characters in S, except those in T

```



Special pattern characters

Many characters in a pattern are taken literally; they are matched during parsing and output unchanged during formatting. Special characters, on the other hand, stand for other characters, strings, or classes of characters. They must be quoted, unless noted otherwise, if they are to appear in the prefix or suffix as literals.

Symbol	Location	Localized?	Meaning
0	Number	Y	Digit
#	Number	Y	Digit, zero shows as absent
.	Number	Y	Decimal separator or monetary decimal separator
-	Number	Y	Minus sign
,	Number	Y	Grouping separator
E	Number	Y	Separates mantissa and exponent in scientific notation. Need not be quoted in prefix or suffix.
;	Subpattern boundary	Y	Separates positive and negative subpatterns
%	Prefix or suffix	Y	Multiply by 100 and show as percentage
‰ (hex 2030, "per mille" sign)	Prefix or suffix	Y	Multiply by 1000 and show as per mille
¤ (hex 00A4 "Currency" sign)	Prefix or suffix	N	Currency sign, replaced by currency symbol. If doubled, replaced by international currency symbol. If present in a pattern, the monetary decimal separator is used instead of the decimal separator.
'	Prefix or suffix	N	Used to quote special characters in a prefix or suffix, for example, "'#'" formats 123 to "'#123'". To create a single quote itself, use two in a row: "'# o'clock'".

Default numeric format

The default numeric format is "0" for integer numbers and decimal numbers with zero digits of precision.

The default numeric format is "0.#" for decimal numbers with at least one digit of precision.

Date and time formatting

The instructions for date formatting are directly out of the Java class `com.text.SimpleDateFormat`¹.

Since dates and times are stored together in a single value, a single substitution formatter can print both or any part of the two. When reading some content source files Exclamation uses the column date format in the Content Declaration file (see "type=date[:format]" on page 19) to determine whether the original content is just a date, just a time, or both a date and time.

Date format string

A format string is a WYSIWYG-like model for selecting and displaying portions of a Date/Time value. This string is comprised of date and time element "tokens" along with any desired spacing or punctuation characters. For example,

Format string:

```
"EEE, MMM d, yyyy 'at' h:mm a z"
```

Sample result:

```
Wed, December 25, 2002 at 5:15 PM EST
```

1. <http://java.sun.com/j2se/1.3/docs/api/java/text/SimpleDateFormat.html>



Date format syntax

To specify the time format use a *time pattern* string. In this pattern, letters A-Z and a-z are reserved as pattern letters, which are defined as the following:

Symbol	Meaning	Presentation	Example
G	era designator	(Text)	AD
y	year	(Number)	1996
M	month in year	(Text & Number)	July & 07
d	day in month	(Number)	10
h	hour in am/pm (1-12)	(Number)	12
H	hour in day (0-23)	(Number)	0
m	minute in hour	(Number)	30
s	second in minute	(Number)	55
S	millisecond	(Number)	978
E	day in week	(Text)	Tuesday
D	day in year	(Number)	189
F	day of week in month	(Number)	2 (second Wednesday in July)
w	week in year	(Number)	27
W	week in month	(Number)	2
a	am/pm marker	(Text)	PM
k	hour in day (1-24)	(Number)	24
K	hour in am/pm (0-11)	(Number)	0
z	time zone	(Text)	Pacific Standard Time
	'escape for text	(Delimiter)	
	''two apostrophes	(Literal)	'

The count of pattern letters determine the format.

(Text):

4 or more pattern letters–use full form, less than 4–use short or abbreviated form if one exists.

**(Number):**

the minimum number of digits. Shorter numbers are zero-padded to this amount. Year is handled specially; that is, if the count of 'y' is 2, the Year will be truncated to 2 digits.

(Text & Number):

3 or over, use text, otherwise use number.

Any characters in the pattern that are not in the ranges of ['a'..'z'] and ['A'..'Z'] will be treated as quoted text. For instance, characters like ':', '.', ',', '#', and '@' will appear in the resulting time text even they are not embraced within single quotes.

A pattern containing any invalid pattern letter will result in a warning diagnostic with the code "BADFMT".

Examples

"yyyy.MM.dd G 'at' hh:mm:ss z"	1996.07.10 AD at 15:08:56 PDT
"EEE, MMM d, 'yy'"	Wed, Jul 10, '96
"h:mm a"	12:08 PM
"hh 'o'clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:00 PM, PST
"yyyyy.MMMMM.dd GGG hh:mm aaa"	1996.July.10 AD 12:08 PM

Default date and time format

When a Date/Time substitution does not include a format specification, Exclamation uses the default. The factory default date and time format is locale-specific.



Factory Default Date/Time Format:

Locale	Date	Time	Date & Time
US	"MMMM d, yyyy"	"h:mm a"	"MMMM d, yyyy 'at' h:mm a"
[others tbs]			



Character set conversions

Exclamation is able to convert among a wide variety of character sets, and does so automatically. Character set processing is applied when reading tab delimited database files, the content declaration file, the template declaration file, and the HTML template file.

Internally, all characters read from these files are mapped into the Unicode universal character set.

On output, the bytes written to generated files are converted from Unicode to the same character set as the HTML template file. Characters outside of the range of the generated file's character set are converted to numeric character entities (&#nnn;). Since URLs must use %xx encoding, and only UTF-8 characters are permitted in URLs, only the low eight bits if the Unicode encoding value are encoded.

How character sets are determined by Exclamation

Some control can be exercised over the character sets used in your files. For all input files, Exclamation looks for a Unicode byte order mark (BOM) at the start of the file. The BOM is either 0xFFFFE or 0xFEFF. If the BOM is not there, the file is assumed to be a subset of Unicode. File is examined for other clues.

XML files must begin with the `<?XML?>` processing instruction. The XML specification is followed, so you can indicate the character set in this processing instruction:

```
<?xml encoding='mac'?>
```

HTML files ought to begin with an HTML tag and `<head>` tag. If the `<head>` element has an encoding META tag, that is used to indicate the character set of the HTML template:

```
<html>  
<head>
```



```
<meta http-equiv="content-type" content="text/html; charset=iso-8859-1">
```

Tab delimited database files can have their encoding specified in the content declaration file. The encoding attribute can be given in the `<tabdatabase>` tag:

```
<tabdatabase name="phone list" encoding="Cp1252">
```

The HTML template can have its character set specified explicitly in the template declaration file. The input tag would have an encoding attribute:

```
<input file="index.html" encoding="mac-roman"/>
```

The generated output files are normally the same encoding as the input file. This can be overridden using the output tag's encoding attribute:

```
<output file="[[widget.id]]/index.html" encoding="iso-latin-1"/>
```

If an input file has no byte order mark or explicit encoding attribute, then the operating system default is used. The default is fine to use, but do not share files across different operating systems unless you explicitly specify encoding attributes.

Character set names

Character set names are registered with IANA so they can have a unique name and number assigned to them. Only the official names or their aliases can be used with the encoding attributes in Exclamation. The case of names is not significant.

See the following RFCs for more information:

- 1700: Assigned Numbers (outdated)
- 2278: IANA Charset Registration Procedures

The IANA Web site has a database of character set names:

```
http://www.iana.org/assignments/character-sets
```



The most common character set names are:

- US-ASCII
- Macintosh
- ISO-8859-1 (alias Latin1)
- UTF-8

Some non-standard encoding names are also supported simply because these are the system defaults in Latin-1 countries.

- MacRoman: Apple Macintosh default Latin set
- Cp1252: Microsoft Windows default Latin set

The best place to find a character set name is in the manual of the program that created the file.

Our recommendation

Your Web pages are more likely to succeed with site visitors if they are encoded in ISO-8859-1. Remember, HTML documents can represent any Unicode character through numeric character entities, so unless your document makes a lot of use of an Oriental character set you should use a character set which any browser can read.

So make "ISO-8859-1" the explicit encoding for your HTML template and generated output files. Your tab delimited database files can stay in the system default character set because Exclamation does the right thing if the encoding attribute is omitted.

CHARACTER SET CONVERSIONS

Character set names

-
-
-
-



Index

Symbols

<column>
type attribute data types 18, 19–22

<content>
attributes 15

<expression> 29

<folder>
attributes 15

<input file> 27

<input>
example 25, 26

<keep> 26, 30
example 30

<omit> 26, 30
example 31

<output file> 27
example 27
loop attribute 28–29

<output>
example 25, 26

<query> 29
example 25, 26

<rowlist> 32
example 25, 26
example<*expression*>
example 26

<segmentlist> 30, 31
example 31

<subtable>
attributes 18

<tabdatabase>
attributes 15

<table>
attributes 17

<template> 27
example 25, 26

<tlb> 41, 42, 52, 53
example 42

[*first*]
in implicit loops 54

A

atfirst 61

atlast 62

attribute
<column> type 18, 19–22
if 43
loop 46
vanish 45

attribute 62

attributes
<content> 15
<folder> 15
<subtable> 18
<tabdatabase> 15
<table>
beforelast 47
between 47

B

beforelast 47
example 48
syntax 48

between 47
example 48
syntax 48

BOM 77

brackets
warning on use 55

bycount= 31

byequal= 31

byte order mark 77

C

- case-sensitivity 55
- character type 14
- column declaration
 - examples 20
- columns
 - defining 17
 - example 17
- constants
 - Exclamation 58
- content
 - sources 9
- content database 13
- Content Declaration
 - date format in 72
 - design 13–14
 - file 9
 - file naming convention 55
 - syntax 14
- content folder 9
- content sources 13
- Cp1252 79

D

- data types 19–21
- database 9, 9–11
 - applications 10
 - formats 11
 - using multiple 11
- date
 - keyword 20
 - examples 20

- See also* time 20
 - date*
 - keyword 20
- date and time 19, 72–75
 - date string 72
 - date syntax 73
 - default 74
 - formatting in Content Declaration 72

- date[:format]* 19

- decimal* 62

- decimal math 66–67

- declaration paradoxes 37

- default date and time formatting 74

- default numeric format 72

- define

- columns 17

- double square brackets

- warning on use 55

E

- engineering notation 69
- Exclamation
 - constants 58
 - expressions 59
 - overview 7–8
 - symbols 58
 - syntax 58–67
- Exclamation *<tlb>* tag 41, 42
- exponent 70
- expression
 - symbols 59
- expressions

- examples 60
 - Exclamation 59
 - functions 59
 - operators 59
 - precedence levels 60
- external type 14

F

- file naming conventions 55
- formatting
 - date and time 72
 - number 68
 - substitution 68
- functions
 - atfirst* 61
 - atlast* 62
 - attribute* 62
 - decimal* 62
 - expressions 59
 - hasnorows* 63
 - hasrows* 63
 - if* 63
 - isok* 64
 - mod* 64
 - numberofrows* 64
 - positionof* 64
 - url* 66

H

- hasnorows* 63
- hasrows* 63
- HTML
 - case-sensitivity 55

- conditional commands 43–46
- support for template files 41
- html 14
- HTML commands
 - if* 43
- HTML elements
 - begin/end elements 56
 - begin-only elements 56
 - Exclamation specific 41–42
- HTML template
 - See also* template file 39

I

- IANA 78
- if* 63
 - examples 43
- implicit looping
 - [first]* subscript 54
 - examples 50, 52
 - loopable tags 51
 - rules for using 51
 - See also* looping 49
 - why it is needed 51
- input decoding 75
- integer digits 69
- ISO 79
- isok* 64

J

- join* 26

K

- keyword
 - date* 20
 - time* 20

L

- Latin set 79
- loop
 - attribute of *<output file>* 28–29
- loop* 46
 - syntax 46
- looping commands 46–54

M

- Macintosh 79
 - character set 79
- MacRoman 79
- mantissa 70
- mod* 64

N

- negative exponents 69
- number formatting 68
 - default 72
 - pattern syntax 70
 - scientific notation 69
 - special pattern characters 71
- number patterns 68
- numberofrows* 64

O

- operators
 - expressions 59
- output encoding 75

P

- paragraph 14
- paragraph text 14
- pattern syntax
 - in number formatting 70
- plaintext 14
- positionof* 64
- precedence levels
 - in expressions 60

Q

- query 26

R

- recursive
 - declaration paradox 37
 - See* recursive
- RFCs 78

S

- scientific notation 69
- See also* looping, implicit
- special pattern characters 71
- square brackets
 - warning on use 55
- substitution 39–41

substitution formatting 68–74
 number formatting 68
 scientific notation 69
 number patterns 68
 syntax 68

subtable
 in Content Declarations 18–19
 syntax 18

subtable queries 32–34
 example 32

symbols
 Exclamation 58

syntax
 beforelast 48
 between 48
 Content Declaration 14
 date and time 73
 Exclamation 58–67
 loop 46
 substitution formatting 68
 subtable 18
 vanish 45
 XML 55–58

T

tab-delimited file 9, 10

tags
 <*tlb*> 53
 See Symbols at the head of the Index

Template Declaration 25
 content 25
 example 25, 35

file naming convention 55

time
 See also date 20

time
 keyword 20

type=
 decimal[:precision] 22
 html 22
 integer 23
 paragraph 23
 plaintext 23
 safetext 23

U

Unicode 77

url 66

URLs 77

US 79

UTF 77, 79

V

vanish 45
 syntax 45

W

wizard icon 7

X

XML 77
 case-sensitivity 55
 XML syntax 55–58