This Document

This document is the second in a series documenting Morris Templating Language (MTL) its uses, usage and syntax.

This document covers "Common" tags - syntax and usage universal to all genres* of MTL. It refers to concepts discussed in "MTL Basic Concepts." If you have not already read "MTL Basic Concepts" or are unfamiliar with the general structure and use of basic programming concepts (like if statements and loops) or don't understand the query structure, you are encouraged to do so.

Conventions

- Unless otherwise noted, this document will show sample results as they would appear in a browser.
- In commentary text, data tags may be discussed in terms of what comes between `<mtl` and `>`, i.e. `<mtl biz.business.id>` may be referred to as "`biz.business.id`" or possibly as "the id tag." In example code, the tags will be fully written out as they should appear in a template.
- "Template," as it will be used in this text, can mean two things: [1] a document in Template Manager (TM) and its contents and/or [2] a section of MTL code that (should) stand alone and work without other supporting MTL code.
- Anything appearing in square brackets: [like this] is just a place holder. Do not insert the square brackets and text inside into a template; replace it with appropriate values.
- `Code to be typed into a template will look like this.`
- Result text (template output) will look like this

Miscellaneous Notes

- This document is not intended to be an exhaustive tag reference. Some sections cover every query qualifier and data tag available in a tag set (at this writing,) others do not. For a full tag reference consult the "MTL Tag Quick Reference Guide."
- Line breaks may be inserted in the middle of MTL tags for clarity throughout this document, but line breaks are NOT LEGAL inside MTL tags**. When/if you copy the code in this document into templates, REMOVE ALL LINE BREAKS that appear between instances of "<mtl" and ">".

About the Author

G. Ann Campbell has been writing MTL since 1999 - when she started by timidly changing the background colors of table cells on onlineathens.com's classifieds templates.

In 2000 she joined MDW and began taking on larger MTL projects, such as the construction of the calendar template set deployed to most Morris Communications Corporation Web sites, and building out classifieds template sets from scratch.

She began teaching MTL in 2001 and started the odyssey of documentation shortly thereafter.

Since then her mission has been to explore strange new tag sets, to seek new uses and new output formats. To boldly go where no MTL-coder has gone before.

*MTL varies slightly depending on the broad subject matter (sports versus classifieds) it's used for.

** Except in the values of variables. It's perfectly okay to put a line break inside a variable:

```
<mtl $okayLinebreak="The following line break:
is fine. It ends up in the value of the variable.">
```

Table of Contents

Getting started

In "MTL Basic Concepts" we only worked with pseudo-MTL. Now it's time to get to the real thing.

You'll be writing your templates in Template Manager (TM.) If you don't have a TM account, contact your Support representative for download instructions and basic training in how to use the application.

But before you start writing templates, you need to be aware that the underlying database Forge pulls from is a shared resource. It houses and services not just your data and queries, but those of many other people too.

So by writing "good" (efficient) templates, you're not just working in your own best interest (an efficient template "comes back" to the browser faster) but being a good citizen. How do you write "good" MTL? Generally, by making your queries as specific as possible. We'll cover that in detail later, so at this point just keep it in mind.

Working in TM

TM is organized hierarchically . There are organizations, and within those organizations there are properties, and under that levels of categories. You have the ability to create categories and sub-categories in your property. If you're working with classifieds (described fully in "MTL Classifieds Tags") your category name must be "CLASSIFEIDS." Otherwise, you're free to name your categories as you see fit. But you should always pick names that describe what's inside: "business directory" rather than "slow Tuesday" for instance.

For the purposes of this document, create a "test" category under your TM property if one doesn't already exist. Under that create a category named after you. (If you don't understand what a property is or how to create a category, contact your Support rep. for training.) This is where you'll put the templates you create as you work through this documentation, creating in effect a mini code library you can refer to later.

First Template:

All that said, it's time to create your first template: Add a template to your sub-category in test.

Generally, you should give templates names that tell something about what the template's supposed to do. Name this one "hello world."

After name, there are five configurable attributes for each template:

- type - The type of your templates should generally reflect the "flavor" of MTL you'll be using, i.e. "biz" for templates that use mainly Biz MTL, "classified" for classifieds, etc.
- Status - should be "active" for any live templates.
- Default - should be set to true for production templates (you can use "false" for templates you're testing.)
- sub-type - should vary according to its function. Sub-types of note are:

`TOC`	this stands for Table of Contents - this is generally used as the first page in a section.
`Search`	A search form/interface
`Browse`	record display
`Detail`	record display
`Alt_Detail`	record display - this is often used for alternate presentation, e.g. a printer friendly version
`Clipboard`	this sub-type is used in conjunction with the 'My List' feature - to display the records the user selected

`Include` you **must** use this sub-type for code (MTL/HTML/plain text) you'll be including into other templates. You cannot include a template that is not of this type

- level - Set this to 1 for the templates you expect to use the most; it's the default level. The original intent of the level input was to distinguish among multi-page interfaces, like data entry forms or iterative search (search within results.) So page 1 of an interface might be at level 1, and the second page at level 2 &etc.

For the template you're creating now, pick "Calendar" for the type (the type for this template doesn't actually matter since there won't be any MTL in it) and "Detail" for sub-type. Set default to "true."

Now click the "template body" tab in your template window. You should be staring at a blank screen. Type:

```
Hello World<br>
```

Save the template.

The fastest and probably easiest way to see the output of this template is to use TM's built-in template previewer (File->Preview Template…, then hit 'OK' on the dialog.) This is a great way to get a quick peek at how your templating's going. There are only two drawbacks: TM's previewer isn't a full browser and tends not to handle complicated HTML well, and your users aren't going to have the option of using it.

Accessing your templates

To get to your template from the Web, a dedicated second or third level domain is required. Typically a third level domain is used. They often take the form "[genre].[something here].com." Genre is usually something like "classifieds" or "calendar" or "sports." If you're not sure what yours is, contact your Support rep.

The URL to access your templates will follow this pattern:

```
http://[domain]/[RPE]?category=[your category]&temp_type=[your sub-
type]&tp=[your property]
```

RPE stands for "request processing engine," a generic name for the programs that take requests from the Web, process and respond to them. There are a few slightly different versions: "classifieds-bin/classifieds," "realestate" and "forge." What's the difference? Not much. They're all names for basically the same thing. Using classifieds-bin/classifieds triggers a slightly different, classifieds-centric, behavior in some situations, and the realestate RPE is a little more tuned to the real estate vertical – an outgrowth of classifieds. We'll be using Forge in this document. (By the way, it's "forge" - all lower case - in a URL, but actually named "Forge")

"Your classification" is going to be the name of the classification your template is stored in. "Your sub-type" will be the sub-type of the template you're trying to get to. Of course "your property" is the name of your TM property.

So in the example above, if the temp_type in your URL is "detail," you'll get back the active, default, level 1 detail template directly under your classification in the specified property.

In any category you can only have one template of a given subtype and level set to status=active, default=true. (i.e. only one default, active, level 1 detail template in a category.) When you need a second template of a given sub-type, set level to 2 and add "&tl=2" (for template level) to your URL:

```
http://[domain]/forge?category=[your category]&temp_type=[sub-
type]&tl=2&tp=[property]
```

While you can't have multiple default, level 1, detail templates directly under the same category, you can have one under your top category, Testing, and one under each of its sub-categories.

The category you specify in your URL tells Forge where to start looking for a template that meets the temp_type and level requested. If it doesn't find one there, it'll go to the next level up and look there (and so on.) We'll talk soon about what happens if it gets all the way up without finding a template with the proper attributes.

When your property's dedicated domain was set up, certain associations may have been made for it. For instance, the classifieds.onlineathens.com domain is associated with the "Athens Ga" template property in TM. So when you go to:

```
http://classifieds.onlineathens.com/classifieds-
bin/classifieds?temp_type=toc
```

You'll get the default, level 1, TOC template in the Athens Ga property. If a similar association was made for your domain you can leave the `tp=` argument out of your URL's. If it hasn't been made, you can contact your Support rep. to have it made, or simply continue using `tp=` in URL's.

In either case, you can use your dedicated domain to access templates from other properties with the tp= argument, like so:

```
http://classifieds.onlineathens.com/classifieds-
bin/classifieds?temp_type=toc&tp=augusta+ga
```

Of course, this is mainly for testing; you don't want to deploy links to another site's third level domain: aside from the branding issues, the other site will get credit for your hits.

Now that you know how to link to your templates the "right" way, here's the quick-and-dirty, only-for-testing way. Look at your template's title bar. The number in parentheses is the Template ID (tid.) Now in a browser go to:

```
http://[domain]/forge?tid=[your tid]
```

Now you have three ways to see your template output.

All three methods will give you the output of your template (although TM's built-in parser is a little slow with complicated html) but the "long way" is what you should use for public links to templates. Why? Because it allows more flexibility in the long run.

Say you develop a template, decide it's wonderful and put tid-style links to it all over your site. Then six months later, you decide to overhaul it. So you start redevelopment in a second template, decide it's wonderful and get ready to deploy the new version. If you've used tid-style links you either have to find every link to the old template and change the tid, or you have to replace the contents of the old template with the new template (losing the ability to reference the old template.)

But if you've used long-style URL's, you set the old template to inactive or its default to false. Then set the new template to the same sub-type and level as the old one and default to true. And you're done.

Exercises:

Ex. 1: Create a long-style URL to the "Hello World" template you created in *Working in TM*.

Redirecting

Missing templates

Getting back to the question of what happens when Forge can't find a template that meets the criteria you specified:

On a Web server you have the ability to customize your "file not found" page. Forge gives you that ability too.

If asked for a template it can't find, Forge will check your property to see if has a "file not found" URL listed (contact your Support rep to have this set up.) If it does, the user will be sent there.

If Forge doesn't find a URL, it'll look at the host name the request was made for. If it's a third-level domain, Forge will try to guess the address of the main site, by stripping off the third level. For example, Forge would whittle `boo.hoo.com` down to hoo.com. Then Forge redirects the user to a page named templatenotfound.shtml at that site, which in our example would be http://hoo.com/templatenotfound.html.

Then either the Web server finds that "we can't find the dynamic content you were looking for" page and serves it to the user, or the Web server doesn't find it, and give the user the site's normal "file not found" message.

Location header

You can also bounce users around on purpose if you want, like this:

```
Location: [new location]
```

But, this must be the first line of the template, and must be followed by a blank line for it to work. Also nothing in the location line will be parsed, so if you do this:

```
Location: http://[domain]/new_path?date=<mtl time.date>
```

The user will end up at:

http://[domain]/new_path?date=<mtl time.date>

Also, template execution is aborted after a location header. So in a template like this:

```
Location: http://[domain]/new_path?date=<mtl time.date>

<mtl loop times=3>
     boo<br>
</mtl loop>
```

Forge wouldn't even get to the loop structure, much less execute it.

Time, time, time

The tags

While most MTL tags are associated with specific sets of data, there are some utility-type tags that are available outside of any given query.

One large set is the time/date tags. An exhaustive list of the tags and their definitions is available in the "MTL Tag Quick Reference Guide," and you can look them up there, so we'll just touch on a few.

You can pull any given piece of a date/timestamp like so:

```
<mtl time.hour>:<mtl time.minute> <mtl time.am_pm> <mtl time.month>/<mtl
time.day>/<mtl time.year>
```

Which gives you the current hour, current minute, current month, day and year:

```
9:41 AM 2/11/2002
```

Of course, it's a good bit less typing to do this:

```
<mtl time.string>
```

Which returns a full time/date stamp:

```
Mon Feb 11 09:41:32 2002
```

But if all you're looking for is the date, it's just as easy to do this:

```
<mtl time.date>
```

Which returns this:

```
2/11/2002
```

To get the date for another day, pass the day or date in to the tag, like so:

```
<mtl time.date(tomorrow)>
```

Which, at the time of this writing was:

```
2/12/2002
```

Similarly, you can pass a day or date into any of the other tags in any of the following (case insensitive) formats. Any date can be followed by a `+[n]` or `-[n]` (for example `TODAY+7`)

Absolute dates:

```
3/17/99
3/17/1999
3-17-99
3-17-1999
17-mar-99
17-mar-1999
19990317
```

Relative dates:

```
TODAY
TOMORROW
YESTERDAY
GAMEDAY (times before 8 am are treated as the previous day)
LAST_[weekday here]
NEXT_[weekday here]
[weekday here]_BEFORE([date])
[weekday here]_AFTER([date])
```

Where "weekday" = `monday tuesday wednesday thursday friday saturday sunday.` Weekdays may be abbreviated down to 2 letters.

All date tags default to "today," but will accept an absolute or relative date as an argument.

So on Tuesday, July 30, 2002, these dates:

```
Sunday_before
Sunday_before(today)
Sunday_before(20020730)
```

All mean the same thing: July 28th. You can't however, use them all in the same way. For more on this, see *Storing & manipulating dates* below.

About weekdays

When you use weekday names to specify dates, they'll be **relative to the current day**. `Thursday` will be the closest Thursday to today. So if today is Friday, then `thursday` means yesterday (i.e. a day in the past,) but if today is Tuesday, then `thursday` means two days in the future. (Head spinning? Take a deep breath. It'll go away.)

The "relativity" of weekday names can be a problem, especially in situations like you find in classifieds, where you may want to offer readers the option to search today's ads and/or Sunday's ads. If you use those specific words (`today`, `sunday`) as your dates, then the `sunday` search option will work until Wednesday. But users who search on Thursday, Friday and Saturday will be trying to search *next* Sunday's ads - which more than likely aren't available yet. Even if they are, they're not ready for release.

What to do? What to do?

Let's say Sunday was the first. This:

```
sunday_before_today
```

Means Sunday, the 1st not just on Monday, Tuesday and Wednesday, but also on Thursday, Friday and Saturday. But what does it mean on Sunday the 8th? It means the Sunday before the current day, Sunday the 8th. So on Sunday the 8th, giving users the search options labeled "today" and "Sunday" (the actual input value here would be `sunday_before_today`, of course) let's them search two consecutive Sundays.

Some people don't like that, tho. So their search inputs have values of `today` and `sunday_before_tomorrow`. Personally, I find this confusing because on Sunday itself, both inputs end up referring to the same day and therefore search the same set of ads.

About months

All date-related tags default to *now*, and will accept an argument in any of the recognized date formats. The exception to this rule is the month-specific tags: `<mtl time.month_name(optional arg)>`, `<mtl time.days_in_month(optional arg)>` and `<mtl time.month_num(optional arg)>`. These tags *do* default to today and *do* accept a full date as an argument. But they will *also* accept a number from 1 to 12:

```
<mtl time.month_name(3)>
```

Returns:

March

And

```
<mtl loop times=12>
     <mtl time.month_name(loop.index)>,
     <mtl time.days_in_month(loop.index)><br>
</mtl loop>
```

Returns:

January, 31
February, 28
March, 31
April, 30
May, 31
June, 30
July, 31
August, 31
September, 30
October, 31
November, 30
December, 31

(For more on using forms with MTL, see *Form-ing* below.)

Storing & manipulating dates

Storing dates

All time-related tags will accept a date argument. You can hard-code the date, like so:

```
<mtl time.date(tomorrow)>
<mtl time.date(yesterday-43)>
<mtl time.date(11/12/2002)>
```

Or store it in a variable:

```
<mtl $adate="yesterday-43">
```

Then you can do all sorts of nifty things with it:

```
<mtl $adate><br>
<mtl time.date($adate)><br>
<mtl time.weekday_name($adate)><br>
```

Here's the output (at this writing):

yesterday-43
6/16/2002
Sunday

When you spit $adate out "straight," you get just what you put in: yesterday-43, but when you pass it through a date tag, it gets treated like a date because it looks like a date: `[date]-n`.

Conversely, this:

```
<mtl $bdate=43>
```

Doesn't look like a date, according to MTL's recognized date formats, so when you do this:

```
<mtl $bdate><br>
<mtl time.date($bdate)><br>
<mtl time.weekday_name($bdate)><br>
```

You get this:

43
Bad Date
Bad Date

Formatting dates

There are tags for just about any part of a time/date stamp, and you can use them to assemble one in just about any format:

```
<mtl $adate=wednesday_before(last_wednesday)>
<mtl time.weekday_name($adate)> <mtl time.day($adate)> <mtl
time.month_name($adate)> <mtl time.year><br>
```

Returns:

Wednesday 17 July 2002

Or, you can just format your variable output:

```
<mtl $adate=wednesday_before(last_wednesday)>
<mtl $adate date_fmt="Day dd Month YYYY"><br>
```

Returns pretty much the same thing:

Wednesday 17 July 2002

You can also use date formatting to do things you can't do very easily with the time data tags:

```
<mtl $adate=wednesday_before(last_wednesday)>
<mtl $adate date_fmt="Dy. dd Mon yy"><br>
```

Returns:

Wed. 17 Jul 02

Notice that I get out all (and only) the punctuation I put in: period after my abbreviated day, but none after my abbreviated month.

Notice also the "date" that I'm using: `wednesday_before(last_wednesday)`. It may not be the first date format that pops to mind, but it's perfectly legal; `wednesday_before()` will accept any valid date as an argument, and `last_wednesday` is a valid date.

Here are the format options available. You can mix and match the formatting characters however you want, put them in any order you want (year then day then seconds or vice versa), use as many or as few as you want, and mix in as many or as few literal characters (punctuation, words, &etc) as you want.:

DATE_FMT uses the following symbols. If the date is September 3, 2001 the following characters are replaced with:

```
dd    = 3
```

```
ddth = 3rd
ddTH = 3RD
DD   = 03
mm   = 9
MM   = 09
yy | YY = 01
yyyy | YYYY = 2001
mon  = sep
Mon  = Sep
MON  = SEP
month = september
Month = September
MONTH = SEPTEMBER
day  = monday
Day  = Monday
DAY  = MONDAY
dy   = mon
Dy   = Mon
DY   = MON
```

TIME_FMT uses the following symbols. If the time is 8:05 PM, the following characters are replaced with:

```
hh   = 8
HH   = 08
mi | MI   = 05
am | pm   = pm
AM | PM   = PM
```

You may be wondering at this point what the difference is between `fmt=`, `date_fmt=` and `time_fmt=`. All three do the same thing: they **modify** your output, but *only* the output from these tags:

- `<mtl time.date>`
- `<mtl time.string>`
- `<mtl $[variable containing a date>` - this works only with `date_fmt=`

The difference among the modifiers is in the kinds of modifications you can use them to do:

- `date_fmt=` Use this for data tags and variables that return dates. (`time.date` and `time.string` are the only Common tags you can apply this to, but other tag sets, including Cal, Biz and Sports, have data tags that return dates.) This modifier flags Forge that it's dealing specifically with a date. Date-formatting a variable *only* works with this modifier.

- time_fmt = Use this to format any data tag that returns a time. `time.string` is the only Common tag this applies to, but other tag sets, including Cal, Biz and Sports, have data tags that return times.
- fmt= You *can* use this modifier to apply formatting to a time/date data tag; Forge already realizes it's dealing with a date/time. But this is *not* preferred.

It bears repeating that you can't apply formatting to any other date or time-related tag than <mtl time.date>, <mtl time.string> or variables with valid dates in them. For instance, we saw before that this:

```
<mtl $adate=Wednesday_before(last_wednesday)>
<mtl $adate date_fmt="Dy. dd Mon yy"><br>
```

Works just fine:

Wed. 17 Jul 02

But if you try this:

```
<mtl $adate=Wednesday_before(last_wednesday)>
<mtl time.weekday_name($adate) fmt="Dy.">
```

You get an error. Because you can't `fmt=[date stuff]` or `date_fmt=` any time/date tag except `<mtl time.date>` and `<mtl time.string>`.

Similarly, if your variable doesn't take a form that Forge recognizes as a date, your attempts to use it in a time tag or with `date_fmt=` will fail:

```
<mtl $adate=4-july-2002>
<mtl $adate date_fmt="Day, Month dd, yy">
```

And:

```
<mtl $adate=4-july-2002>
<mtl time.date($adate) date_fmt="Day, Month dd, yy">
```

Both come back with errors - because Forge doesn't see "4-july-2002" as a date; it only recognizes the month abbreviations, not the full month names.

By the way, both versions of the above output attempt are functionally the same. And if we format the date in a way Forge recognizes:

```
<mtl $adate=4-jul-2002>
<mtl $adate date_fmt="Day, Month dd, yy"><br>
<mtl time.date($adate) date_fmt="Day, Month dd, yy"><br>
```

They both work just fine, and give the same result:

Thursday, July 4, 02
Thursday, July 4, 02

Date arithmetic

You can format a variable containing a date in any of the date formats specified above - including `[date]+[n]`:

```
<mtl $adate=4-jul-2002>
<mtl loop times=7>
     <mtl $adate={$adate}+1>
     <mtl time.date($adate) date_fmt=Day><br>
</mtl loop>
```

Note the curly braces in the second variable assignment - without them you get an error telling you "$adate+1" is an invalid variable assignment:

Friday
Saturday
Sunday
Monday
Tuesday
Wednesday
Thursday

We could also do it this way:

```
<mtl $adate=4-jul-2002>
<mtl loop times=7>
     <mtl {$adate}+{loop.index} date_fmt=Day><br>
</mtl loop>
```

Note the curly braces around `$adate` and `loop.index`. Instead of adding one to the variable and storing the new total in `$adate` each time, this version has Forge do the math at output. The result is the same:

Friday
Saturday
Sunday
Monday
Tuesday
Wednesday
Thursday

But be aware that the two mechanisms are slightly different. When you do this:

```
<mtl $adate=4-jul-2002>
<mtl loop times=4>
     <mtl $adate> <mtl loop.index>:
```

```
     <mtl {$adate}+{loop.index} date_fmt=Day><br>
</mtl loop>
```

The value in `$adate` never changes, even though the day that's output does:

4-jul-2002 1: Friday
4-jul-2002 2: Saturday
4-jul-2002 3: Sunday
4-jul-2002 4: Monday

But this version does change the value of `$adate`, although probably not in the way you anticipated:

```
<mtl $adate=4-jul-2002>
<mtl loop times=4>
     <mtl $adate={$adate}+1>
     <mtl $adate> : <mtl $adate date_fmt=Day><br>
</mtl loop>
```

Returns:

4-jul-2002+1 : Friday
4-jul-2002+1+1 : Saturday
4-jul-2002+1+1+1 : Sunday
4-jul-2002+1+1+1+1 : Monday

Why? Because we've used a concatenation syntax*: we've told Forge to evaluate (that's what the curly braces do) `$adate`, then tack "`+1`" on to the end.

Despite the fact that the value of the variable in the example above looks rather strange, when you perform date operations on it (pass it into a time tag or use `date_fmt=`) Forge reads it just like it reads Today+4: as a date value.

Forge's date addition even works across month and year boundaries:

```
<mtl $adate=26-feb-2002>
<mtl loop times=4>
     <mtl $adate={$adate}+1>
     <mtl $adate> : <mtl $adate date_fmt="Mon. dd"><br>
</mtl loop>
```

Returns:

26-feb-2002+1 : Feb. 27
26-feb-2002+1+1 : Feb. 28
26-feb-2002+1+1+1 : Mar. 1
26-feb-2002+1+1+1+1 : Mar. 2

And

```
<mtl $adate=29-dec-2002>
```

```
<mtl loop times=4>
     <mtl $adate={$adate}+1>
     <mtl $adate date_fmt="Mon. dd yyyy"><br>
</mtl loop>
```

Returns

Dec. 30 2002
Dec. 31 2002
Jan. 1 2003
Jan. 2 2003

* If we'd used addition syntax: `<mtl $adate={$adate+1}>` the result would not have been a valid date.

Testing time

Often you'll need to test one date against another . Here's the syntax for that:

```
<mtl if date=today eq=Thursday>
<mtl if date=17-mar-1999 before=Friday>
<mtl if date=$adate after=YESTERDAY>
```

You can use any valid date format (see above) in either comparison slot. Remember that days of the week are relative. So this:

```
<mtl if date=thursday ne=1/1/2003>
```

Does not check to see that the first day of 2003 will be (or was) a Thursday. It checks to see whether the closest Thursday to today is the first day of 2003.

Note the use of `date=` rather than `val=` in all the above if statements. `Date=` flags Forge to do date calculations. But this:

```
<mtl if val=today eq=Thursday>
```

Would compare the word "today" to "Thursday," and since the words are different, come up with "false" - even if `today` actually is Thursday.

Exercises:

You can do the following exercises in a single template if you like.

Ex. 1: Get the dates for tomorrow, next Thursday and 48 days from now.

Ex. 2: Use the tags to see what day of the week your birthday will be on in 2030.

Ex. 3: Combine time tags with a loop to find the weekday of your birthday for every year from now to 2030.

Ex. 4: Use a loop to output the dates for your birth month. Use a date test to bold your birthday.

Modifying your output (some more)

You've seen already how to modify the output of time/date tags, but you can modify the output of other tags (data tags and variables) too. These modifiers work on all data tags:

`blanks=[x]`	`x` is returned if the value of the data tag or variable is blank
`blanklines=[x]`	`x` is used to replace blank lines in the string
`case=upper`	output tag IN ALL UPPER CASE
`case=lower`	output tag in all lower case
`case=title`	Capitalizes the First Letter of Each Word Except Things Like and, the, a and of.
`case=initcap`	Capitalizes The First Letter Of Each Word.

escape_html=yes	Changes '<' to < &etc. so that text containing html tags can be displayed verbatim in browsers (without having to use <pre>.) For example, this:
	
	`<mtl $string='<a href="http://yahoo.com">&amp;Boo, yahoo!</a>'>`
	`<mtl $string escape_html=yes>`
	
	Shows up in a browser as this:
	
	&Boo, yahoo!
	
	Look at the source, and you'll see this:
	
	&Boo, yahoo!

`fmt=.2`	round your number to the specified number of decimal places
`fmt=url`	URL-encodes the tag output (translates spaces, dashes &etc.) This can, however, have unintended consequences: `<mtl $test="083">` `<mtl $test fmt=url> ` `<mtl {$test} fmt=url> ` `<mtl {$test fmt=url}> ` Gets you this: 083 083 83 Normally, "083" and "83" mean basically the same thing, but

	sometimes - like when you're dealing with classifieds' category numbers - they don't. A more reliable alternative is the string.url_encode() function (for more on this, see *Fun with Strings*, in "MTL Basic Concepts") - which will perform the same formatting without otherwise changing your values
`linebreaks=[x]`	`x` is used to replace line breaks in the string (use this to automatically convert returns in your text to 's or <p>'s &etc. in your output)
`max_chars=[n]`	`n` should be a number. The returned string will be no more than `n` characters long. Elipses (…) will be appended if there were more than `n` characters in the string.
`strip_chars=[x]`	`x` is a comma-delimited list of characters to be stripped out of the output.
`strip_html=yes`	strips html from output

There are data tags that have modifiers that apply specifically, and only, to them. (For an example, see page_nav_bar in *Pagination*.) The ones above apply to all data tags.

You can use multiple modifiers in a given tag. For instance:

```
<mtl $test="capitalize the first letter of each word except things like
the and a and of.">
<mtl $test case=title strip_chars='t,T'>
```

Returns:

Capialize he Firs Leer of Each Word Excep hings Like he and a and of.

Notice that "things" became "hings." The 't' got capitalized by `case=title`, but then the capital 't' was stripped off by the `strip_chars` modifier. But if we mix it up some:

```
<mtl $test="capitalize the first letter of each word except things like
the and a and of.">
<mtl $test strip_chars='t,T' case=title>
```

"things" becomes "Hings.":

Capialize He Firs Leer of Each Word Excep Hings Like He and a and of.

Because the modifiers got applied in the order they appeared in the tag. So in the first example Forge applied `case=title`, then stripped upper and lower case 't's. In the second example the order was reversed, so when the 't' got stripped off of "the," 'h' became the first letter of the word and it got capitalized.

Variables from 'the outside'

At this point, you know how to create a variable and give it a value:

```
<mtl $var="boo">
```

And how to get that value back out:

```
<mtl $var>
```

But if your variables are all hard-coded into your templates, they're of limited use. Fortunately, it is possible to set a variable outside your template and pass it in.

When you made your first template, you accessed it like this:

```
http://[domain]/forge?tid=[your tid].
```

And like this:

```
http://[domain]/forge?category=[category
name]&temp_type=detail&tp=[property name]
```

And via TM's preview feature.

In all three cases your access followed this pattern:

```
http://[a listener]/forge?[query string]
```

Everything after the question mark in a URL is the query string. When you access Forge, the first thing it does is grab the URL you got there with, and "break off" everything after the question mark. That's the "query string."

So out of this:

```
http://[your listener here]/forge?category=[category
name]&temp_type=detail
```

Forge extracts this:

```
category=[category name]&temp_type=detail
```

The next step is to chop the query string at the ampersands (&) into chunks.

So out of this:

```
category=[category name]&temp_type=detail
```

Forge gets this:

```
category=[category name]
temp_type=detail
```

The last step is to split the chunks into variables and values, so Forge takes this:

```
category=[category name]
temp_type=detail
```

And translates it to something like this:

```
<mtl $category=[category name]>
<mtl $temp_type=detail>
```

Note that this MTL code doesn't appear anywhere. Forge sets these variables "behind the scenes" for you and acts as though you had written the code that appears above.

So after all that discussion, let's wrench ourselves back to the question at hand: how do you set a variable outside your template?

And at this point the answer is hopefully obvious: add it to the end of your query string. So this URL:

```
http://[your listener here]/forge?category=[category
name]&temp_type=detail
```

Becomes this:

```
http://[your listener here]/forge?category=[category
name]&temp_type=detail&var=boo
```

Notice, that there's no dollar sign ($) in this variable assignment; just the variable name and the variable value. Forge takes care of all the rest, and you can tack as many `variable=value` pairs as you want onto a URL as long as they're separated by &'s.

Browser Notes

- You'll need to make sure you URL-encode any "special" characters (like spaces, punctuation and other non-letter characters) in your variable values. Netscape 4.*, in particular, will barf on any value that contains a space. Why only URL-encode your variable values? Why not URL-encode variable names too? You shouldn't need to; special characters aren't legal in variable names.
- When writing href's it's recommended that you use the proper format: `&` for the ampersands between arguments, rather than `&`. True, `&` *usually* works, but you can't count on it. Especially when you have variables with names that start with things like "sect" or "reg" or "copy."

For instance, say you have a variable named "sectionID." If you pass it through a url (`&sectionID=4`) Many browsers will be "helpful" and interpret the first part as § (note the additional semicolon) which is this character: §. How to get around this? Make sure your argument-separating ampersands are `&`.

Exercises:

Ex. 1: Modify the template you wrote for Time Ex. 2 to use a variable for the year in question. Call the template from the web, passing in the value of the variable in the URL. Try this with several different years to verify that it works.

Ex. 2: Further modify your template from Ex. 1, above. If there's not a year passed in to the template from the URL, your template should deal with it gracefully by using a default value for year.

Form-ing

Now let's look at how to create a form that submits to your template.

You know that links into templates follow this format:

```
http://[your listener here]/forge?category=[category
name]&temp_type=detail&var1=value1&var2=Stuff%20here&etc=so+on
```

How do you translate that into form format? First, the form tag:

```
<form action="http://[your listener here]/forge" method=get>
```

The form action will be everything that would be to the left of the question mark if you were writing a full URL. **Do not include the question mark in the action**; the browser will supply that for us when the form is submitted.

Use the "get" method; it sends all the form information in the URL like we're used to seeing. Method=post does work, but unless you're writing data entry templates it's not the preferred method.*

Now the stuff to the right of the question mark. You can use any type of form input. I've used a variety below for illustration:

```
<input type="text" name="category" value="[category name]">
<input type="hidden" name="temp_type" value="detail">
<select name="var1">
     <option value="value1">value1</option>
     <option value="boo">boo</option>
</select>
<textarea name="var2">Stuff here</textarea>
```

Any type of form input works, but of course, you'll want the values your users shouldn't be monkeying with in hidden inputs. When the form is submitted, all these inputs will be translated to name/value pairs:

```
category=[category name]
temp_type=detail
var1=value1
var2=Stuff%20here
```

Notice, the browser does the url-encoding for you.

Then it strings the pairs together:

```
category=[category name]&temp_type=detail&var1=value1&var2=Stuff%20here
```

And joins them to the action with a question mark:

```
http://[your listener
here]/forge?category=calendar&temp_type=detail&var1=boo&var2=Stuff%20here
```

And at this point, it should be back in a recognizable form, yes?

Exercises:

Ex. 1: Create a form (in a static file or in a template - your choice) that submits a user-chosen value to the template you wrote for the *Variables* exercises.

* Why is post the preferred method only for data entry? Because pages requested using the get method are cached to enhance performance, while pages requested with post are not.

So the get method makes the most sense for normal data retrieval pages - after all, why request the data from the database and format the results more than once a day when the data only changes once a day? It makes sense to save the results from the first request and serve them up again when you get the second request for the same data.

In addition, the get method puts all the request criteria in the URL, which means less detective work is required when you're debugging a problem: you can look at all the form input right there in the URL.

Including files

So far your templates have been self-contained, but it can be very useful to include code (HTML and/or MTL) from other sources.

You can pull one TM template into another like this:

```
<mtl include name="[name of template]">
```

If the template you're including is located inside your template property, you're done. But if you're trying to include a template from another TM property, you'll need to provide the name of the property, like so:

```
<mtl include name="[name of template]" tp="[name of TM property]">
```

Try it out; create a new template and type:

```
This is a test of including:<br>
<mtl include name="hello world" tp="[your property here]">
This concludes the test.
```

Now save and preview your template. You probably got an error. Why? Because you can only include templates where the sub-type is "include." So make that change to your "hello world" template (from *Working in TM*) and try again. This time you should get:

This is a test of including:
Hello World
This concludes the test.

If you didn't, double check that your template name and the name in the include tag match up (check for leading and/or trailing spaces) and that your template property is spelled correctly.

The template you included happened to only have text in it, but it could just as well have had more MTL.

You can also pull static files (full of MTL, HTML, XML or any combination) from a (Morris) file system into your templates like so:

```
<mtl include file="[domain]:/web/[domain]/htdocs/path/to/file">
```

Although `[domain]` appears twice in the address, what you type may not be the same both times.

For the first instance of domain you'll type the domain name as you would if you were going to log in to the server with an FTP or Telnet client; it tells Forge which box to go to.

Once Forge gets to the box, it needs to know what path to follow. After the colon you type the server-path to the file, and if your build-out is named differently than your domain name (and sometimes they are) you'll type your build-out name for the second instance of [domain].

Give it a whirl by adding this include to your template:

```
<mtl include file="[yourdomain]:/web/[yourdomain]/htdocs/index.shtml">
```

Then save and preview your template (you may want to use a browser: the html of an index page is likely to overwhelm TM's previewer.)

Be aware that none of your server-side includes will have been included: Forge didn't pull the page in through the web server (which is what includes server-side includes) it did a file system operation.

Making files

Output tags

Now that you can include a static file into your templates, let's look at outputting your templates to static files in your (Morris) file system.

Output tags, like query tags and loop tags come in pairs. And everything between the open and close tags will be written out to your file. The format should look familiar; it goes like this:

```
<mtl output file="[domain]:/web/[domain]/path/to/file">
     [stuff to put in the file]
</mtl output>
```

Let's try it out. First, create a test directory in your site's file system, then create a new template:

```
hello<br>
<mtl output file="[domain]:/web/[domain]/[path_to_test_dir]/test.html">
     <mtl loop times=4>
          <mtl loop.index><br>
     </mtl loop>
</mtl output>
goodbye<br>
```

Now preview your template or go to it in a browser. Here's what you get:

hello
1
2
3
4
goodbye

Now check your file system. You'll see that even though your template has output tags, they were not executed and your file was *not* created. Why? Because `<mtl output>` is a specialized tag. Or rather, it's only executed under special circumstances.

When a template is executed from the web (and TM's preview feature simulates a web request) output tags are ignored. When a template is run from the command line, output tags are honored. So, how do you run a template from the command line?

Use your telnet client to log in to your web site. Then type:

```
forge [your tid]
```

You should see this:

Writing: [domain]:/web/[domain]/[path_to_test_dir]/test.html
$

Check your file system. Your file has been written. Go to it in a browser and here's what you'll see:

1
2
3
4

Why is this version of the template output different from the first version? When you call the template from the Web, it's sent to the browser in it's entirety. When it's output to file though, only the stuff between the open and close output tags gets put in the file.

Now, let's try another variation. Delete your test directory and try running the template again.

You get back an error, right? Forge can't output to the path you specified because the path doesn't exist. The final directory is missing.

So make this change to your open output tag and run the template again:

```
<mtl output file="[domain]:/web/[domain]/[path_to_test_dir]/test.html"
mkdir=yes>
```

Forge will output your file without complaint this time. Why? Check your file system and you'll see that it created the missing directory first, then wrote the file. This modifier will make Forge create as many nested directories as it needs to be able to "find" your output path. That is, assuming you have permissions to create them – Forge creates these directories "as" the account that invoked it.

You already know how to pass variable/value pairs into a template you're running from a browser, but how do you pass variables in to templates you're running from the command line?

The method is similar, but without the &'s. This command:

```
forge [your tid]
```

Becomes this command:

```
forge [your tid] var=boo
```

Note that you'll need to put quotes around your value if it contains spaces or other special characters:

```
forge [your tid] var="boo hoo!"
```

Last-minute output with -o

What if you need to output to file a template that doesn't have output tags in it? You can edit the template to add them, or you can do this:

```
forge [tid] -o [path to file]
```

The -o operator tells forge to output the template to file anyway. Forge will attempt to put the file wherever the next command-line argument indicates.

You've got two options for filling in the path to file; you could do a full path from box root, similar to the pathing information required by output tags, or you could supply a path relative to where you are when you output the file. That is, if you've cd'd to the directory you want the file to be created in, then your path to file could just be [filename].[extension].

As you may already have intuited, this:

```
forge [tid] -o var="boo!" [path to file]
```

Just won't get it. Forge will try to name the file 'var="boo!"' rather than what you intended, and try to turn your path into a variable.

Mkdir=yes is not an option in this situation. But why would it be? You're already there. Just make the directories yourself.

You can put "-o [path to file]" anywhere in the template arguments, so this:

```
forge [tid] -o [path] var="boo!"
```

And this

```
forge [tid] var="boo!" -o [path]
```

Will work equally well - as long as the -o flag, and the path aren't separated.

<mtl output> vs. -o

What if you need to output your template to two places? Can you use `<mtl output>` tags to get one file *and* the -o flag to get the other? Nope. If you try it, Forge will tell you it's outputting both files, but when you check your file system, you'll see that only the file specified in the `<mtl output>` tag was created.

But let's say you still need that template output to two different files. Can you put multiple, comma-delimited paths after the -o flag, like so:

```
forge [tid] -o [path1],[path2]
```

Nope, again. If you need two copies of the file, you'll have to run Forge twice. Like this:

```
forge [tid] -o [path1]
```

```
forge [tid] -o [path2]
```

You can, however, put multiple, comma-delimited paths in your <mtl output> tag, like so:

```
<mtl output
file="[domain]:/web/[domain]/[file1],[domain]:/web/[domain]/[file2],">
     [template guts]
</mtl output>
```

Or, assuming your `<mtl output>` tag is set up on some variation of this theme:

```
<mtl output file="[domain]:/web/[domain]/{$file}">
     [template guts]
</mtl output>
```

Run Forge like this:

```
forge [tid] file="[path1]"
forge [tid] file="[path2]"
```

There's one more thing to keep in mind when using the -o flag for output: your file will have the entire output of the template in it.

When you code `<mtl output>` tags into your template, you get to pick where the output starts and where the output ends, as we saw with this example:

```
hello<br>
<mtl output file="[domain]:/web/[domain]/[path_to_test_dir]/test.html">
     <mtl loop times=4>
          <mtl loop.index><br>
     </mtl loop>
</mtl output>
goodbye<br>
```

When you run this template from the command-line, here's what ends up in your file (as seen from the Web):

```
1
2
3
4
```

But if we remove the output tags:

```
hello<br>
     <mtl loop times=4>
          <mtl loop.index><br>
     </mtl loop>
goodbye<br>
```

What ends up in a file generated with the -o flag will be the same as what you'd get by going to the template in a browser:

```
hello
1
2
3
4
goodbye
```

More file-ability

Testing file existence

Occasionally you'll want to know if a file exists before you try to do something with it. For instance, if you try to include a file that doesn't exist, Forge returns an error and stops rendering the template. Or you may want to check that your image exists before you write an image tag for it.

To do that use this syntax:

```
<mtl if file_exists="[domain]:/web/[domain]/path_to_file">
     [stuff here]
</mtl if>
```

Conversely, use this syntax to see if a file doesn't exist:

```
<mtl if file_noexists="[domain]:/web/[domain]/path_to_file">
     [stuff here]
</mtl if>
```

Like all the other operations you've seen where the path starts with "[domain]:/web" this is not a web request, but a file system operation and will work only with files on Morris servers.

You could use this method to make sure you don't have any bad links in your pages:

```
<mtl if file_exists="[domain]:/web/[domain]/htdocs/sports/index.shtml">
     <a href="http://[domain]/sports">Sports</a><br>
</mtl if>
<mtl if file_exists="[domain]:/web/[domain]/htdocs/news/index.shml">
     <a href="http://[domain]/news">News</a>
</mtl if>
[&etc.]
```

Conditional linking with <mtl a>

But in addition to being a **whole lotta** typing, it's also rather cumbersome. Here's the easy way:

```
<mtl a root="[domain]:/web/[domain]/htdocs/"
href="http://[domain]/news/index.shtml">News</mtl a>
```

If your file exists, you get:

```
<a href="http://[domain]/news/index.shtml">News</a>
```

And if it doesn't, you get:

```
<!-- mtl a href="http://[domain]/news/index.shtml" -->News<!-- /mtl a -->
```

If the link target doesn't exist, Forge still outputs the <a> tags, but commented and with the "mtl" left in. Why?

Because Forge keeps a list of "failed" mtl a's in documents it writes out to static files. And if Forge later creates the document you tried to link to with an mtl a (but didn't because the file didn't exist at the time) - Forge recognizes that, goes back to the first document, uses the "mtl" part of the commented <a>'s to find them and *uncomments* the link tags. That's right, Forge not only keeps your links accurate at time of output, but keeps track of the "bad" ones and adds them back in when they become "good."

To suppress the link text when your link target doesn't exist, add "link=skip" to your tag:

```
<mtl a root="[domain]:/web/[domain]/htdocs/"
href="http://[domain]/news/index.shtml" link=skip>News</mtl a>
```

If your file doesn't exists, you get:

```
<!-- mtl a href="http://[domain]/news/index.shtml">News -->
```

Again, if Forge later generates the missing file, it will "correct" the file it output earlier, restoring the link and link text.

Now let's look at the elements of <mtl a>:

```
<mtl a root="[domain]:
```

The first instance of "[domain]" is the hostname. If you're using <mtl a> in a dynamic template or if you're linking to files in a different (Morris-hosted) domain from a template intended to generate static files, hostname is required. If you're generating static files with links to other files in your file system, the hostname is not required, but it doesn't hurt to have it.

```
<mtl a root="[domain]:/web/[domain]/htdocs"
```

After the first [domain]: is the path from box root to the web files directory at the site you're linking to. The first directory in this path must be "web." The third must be "htdocs."

The requirements for the href part of your link are pretty standard:

```
<mtl a root="[…]" href="http://[domain]/news/index.shtml"
```

Paths from dynamically generated pages to static pages need to be full. Paths from static pages to other static pages can be relative. In either case, make sure your "root" path goes up to where your link path starts, and that the two don't overlap each other.

Good:

```
<mtl a root="/web/[domain]/htdocs/sports/" href="boo.txt">Boo</mtl a>
```

```
<mtl a root="[domain]:/web/[domain]/htdocs/"
href="http://[domain]/sports/boo.txt">Boo</mtl a>
```

Bad (notice the overlap):

```
<mtl a root="/web/[domain]/htdocs/sports" href="/sports/boo.txt">Boo</mtl
a>

<mtl a root="[domain]:/web/[domain]/htdocs/sports"
href="http://[domain]/sports/boo.txt">Boo</mtl a>
```

If your link is relative, Forge stitches the href onto the end of root to find the path to the file. If your link is full, Forge chops http://[domain] off the front of the href, then stitches the rest onto the end of root to find the path to the file.

Exercises:

Ex. 1: Modify one of the templates you wrote in the time exercises to output a file to your file system.

Add <mtl a> tags linking to a document you know does *not* exist in your file system. Test the template from the web. Then run the template so that it outputs the file. Verify that the file was created.

Ex. 2: Write a template that outputs to the (missing) file you linked to in exercise 1. Run the template so that it outputs the file. Verify that the file was created. Then take another look at the file you created in exercise 1.

Ex. 3: Write a template to include the file you output in exercise 1, and another template from TM.

Output to email

The program Forge uses to send email is configurable by your System Administrator. This chapter will assume you're running under the default configuration, which is for Sendmail. Consult documentation for the mail engine you're using.

You know by now how to output to the Web, and to a file. There's one more option: email. When you use this option Forge parses your template like normal, but instead of sending the results to a browser or to a file, it sends the results to Sendmail, a common server-side program used to send preformatted messages over the Internet. (When cgi scripts like FormMail.pl send email, they assemble and format the message, then use Sendmail for the actual network routing.)

For email output you use the same `<mtl output>` tags you use for writing to a file, but with email-specific modifiers:

- `email_to=` The address to send email to. May be one or more, comma-delimited, addresses.
- `email_from=` The address the email will be sent from. Leave it blank and it will automatically be filled in to reflect the information (login and first and last names if they're available) of the account used to generate the file.

  Specifically, if you log in as "tester" and run an email-output template without this argument (and without a From: argument in the body of the email, which we'll cover in a minute,) the email that's generated will be labeled as being from "tester@[name of server you logged in to]." If the template is running from a cron, it will be generated "from" the account that owns the crontab.
- `email_subject=` The subject line for the email.
- `email_content_type=` Use "text/html" with this to send HTML email.
- `email_flags=` The Sendmail flags to be used when your email is sent. It's recommended that you put `"-Odeliverymode=q"` in your list of values. It tells Sendmail to queue the emails, rather than trying to send them immediately. By queuing outgoing messages, you alleviate continuous congestion in the mail engine/server. Sendmail will, according to its configuration, flush (send everything in) its queue periodically. email_flags=`-Odeliverymode=q` is particularly recommended for templates that send many messages.

  Leave `email_flag=` blank, and `-t` will be assumed. Specify `email_flag=` values, but leave `-t` out, and it will be added anyway. `-t` tells Sendmail to read email headers from the body of the email. We'll discuss why this is needed in a minute.

Email output is triggered by the use of at least one valid `<mtl output>` modifier that starts with "email_," like so:

```
<mtl output email_to=your_mom@motherhood.com>
Hi Mom!
</mtl output>
```

Run this template from the command line and the email it sent would go to the specified address. Because a from address isn't specified, it'll be labeled as being "from" [Fname] [Lname] <[yourlogin]@[servername]>.

If you want the email sent from a different address, you have two choices:

```
<mtl output email_from=[from address] email_to=your_mom@motherhood.com>
Hi Mom!
</mtl output>
```

Or

```
<mtl output email_to=your_mom@motherhood.com>
From: [from address]

Hi Mom!
</mtl output>
```

Both methods have the same outcome/output. The second method works because Forge is sending a `-t` flag to Sendmail (whether or not you specify it with `email_flags=`), which tells Sendmail to look in the message itself (what's between the `<mtl output>` tags) for header information. Note the blank line in the second version between the last line of header data and the message. It indicates to Sendmail where the headers leave off and the actual message begins. It's required when you use the "look in the body" method, but you can skip it if you're not putting any header information between the `<mtl output>` tags.

Being able to put headers in the body of the email is useful because, as you may have noticed, there isn't a `<mtl output>` modifier for every type of header argument you might want to use. CC and BCC are very good examples. No matter where you put your 'to' and 'from' addresses, this is your only choice for cc-ing and bcc-ing:

```
<mtl output email_from=[from address] email_to=your_mom@motherhood.com>
cc: your_dad@dadhood.com
bcc: your_siblings@childhood.com

Mom,
About those wire hangers…
</mtl output>
```

For a full discussion of email headers, consult Sendmail documentation.

Whether you put your header data in the `<mtl output>` tag or the body of the email doesn't really matter to the outcome, but you have to have at least one `email_[x]=` modifier in your tag to trigger the email. It doesn't really matter which one it is, though. For instance, this works:

```
<mtl output email_flags="-Odeliverymode=q -t">
To: your_mom@motherhood.com
From: a_box@address.com

Dear Mom,
For some reason I want to put all my headers in the body of the message.
</mtl output>
```

And so does this:

```
<mtl output email_flags="">
To: your_mom@motherhood.com
From: a_box@address.com

Dear Mom,
For some reason I want to put all my headers in the body of the message.
</mtl output>
```

On the other hand, there is an advantage to putting at least the to-address in the `<mtl output>` tag. When this template is run:

```
<mtl output email_to=an_address@site.com>
Subject: Hi

Here's the story
of a lovely lady…
</mtl output>
```

You get this confirmation back on the command-line:

```
Sending email to :an_address@site.com
```

And if you put multiple, comma-delimited addresses in the <mtl output> tag:

```
<mtl output email_to=an_address@site.com,another_address@site.com>
Subject: Hi

Here's the story
of a lovely lady…
</mtl output>
```

Gets you this confirmation:

```
Sending email to :an_address@site.com,another_address@site.com
```

"What good is that," you ask?

Well, let's say your template actually looks like this (note the lack of indenting. In this one case I omit it because the indents would show up in my email - which would be a bad thing):

```
<mtl category.table [qualifiers]>
<mtl output email_to=category.table.email_address
email_from=you@yoursite.com>

Dear <mtl category.table.full_name>

Blah, blah, blah.

Sincerely,
Bruno
</mtl output>
</mtl category.table>
```

When you run the template, forge will respond with something like this:

```
Sending email to :addy1@domain.com
Sending email to :addy2@domain.com
...
```

Very handily giving you a record of who you've sent email to. If you're running the template in person (versus running it from a cron) you can save the telnet session to a file. And if you are running the template from a cron, the cron-success email will contain that same list. (Assuming the success message wasn't suppressed when the cron was set up. Contact your Support rep. if you have questions about this.)

Exercises:

Ex. 1: Write a template that outputs to email. Send yourself an email with it.

Output precedence

It might be tempting to kill two birds with one … erm … template, like this:

```
<mtl output email_to=[an_address] email_subject="Hi" file=[path to file]>
Here's the story of a lovely lady…
</mtl output>
```

But an `<mtl output>` tag with a `file=` modifier *and* email modifiers will *only* output the file.

If you call a template from the Web, <mtl output> tags are ignored. If you output a template from command line here's the precedence:

```
file=
email_[X]=
-o
```

So this template:

```
<mtl output file=[path] email_to=[addy]>
[rest of template…]
```

Will *only* output the file. It will not send the email.

This template:

```
<mtl output email_to=[addy] email_subject=[subject]>
[rest of template…]
```

Run with this command:

```
forge [tid] -o [file path]
```

Will *only* send the email. It will not write a file

This template:

```
<mtl output file=[file1]>
[rest of template…]
```

Run with this command:

```
forge [tid] -o [file2]
```

Will *only* output "file1" **even though** Forge may tell you it's writing both files.

Debug & print

Printing values to the command-line

Unfortunately, things don't always go as you planned. Sometimes your templates just won't act right.

And the error messages you get back (if you get any) aren't always helpful.

Of course the first things to check are that your open query tag(s) and close query tag(s) match and that your data tags match your query. Spelling counts!

If you're still having trouble, it can be helpful to sprinkle your template with extra output. I.e. you know what you *think* is in a given variable or data tag at any given time, but verify it. Just for development and testing, go ahead and clutter your output with your variables and data tags so that you can *see* what's in them. (You can clean the template back up before you put it into production.)

Like Ronald Reagan said, "trust, but verify."

If you're running your template from the web doing that is easy. It's just a matter of adding extra output tags.

But if you're running your template from the command line, it becomes a big headache to add the tags to your template, run the template, then go check the file in your file system to see what you got.

There is a better way. Add `print [some label]=` to your data tag, like so:

```
<mtl output file="[domain]:/web/[domain]/[path_to_test_dir]/test.txt">
     <mtl loop times=7>
          <mtl set.date $date=today+{loop.index}>
          <mtl $date date_fmt="Dy."><br>
<mtl print index=loop.index>
     </mtl loop>
</mtl output>
```

On the command line you get this back:

```
Writing: [domain]:/web/[domain]/[path_to_test_dir]/test.txt
INDEX='1'
INDEX='2'
INDEX='3'
INDEX='4'
INDEX='5'
INDEX='6'
INDEX='7'
$
```

And your file looks like this:

```
Fri Mar 22 01:00:00 2002<br>
Sat Mar 23 01:00:00 2002<br>
Sun Mar 24 01:00:00 2002<br>
```

```
Mon Mar 25 01:00:00 2002<br>
Tue Mar 26 01:00:00 2002<br>
Wed Mar 27 01:00:00 2002<br>
Thu Mar 28 01:00:00 2002<br>
```

When you're running a template with these print statements from the web, the tags with "`print [label]=`" are ignored, just like output tags are ignored when you run a template from the web.

When you run the same template from the command line, the data tags you've added the print command to are output only to the command line. Those data tags *are not* output to your file.

Hopefully you noticed that the debug tag, `<mtl print index=loop.index>` was "out-dented." As a rule, I out-dent debug lines so I can find them easily when I'm ready to clean them out and put a template into production. Of course, this only works if the rest of your code is indented.

Debug

Sometimes, after all that, you still have trouble pinning down what's going on. If you're working with a dynamic template, you can get a little more information about what's going on by adding this to your url:

```
&debug=1
```

How much extra data you get back varies depending on what you're doing, but at the very least, you'll be told the id of the template that's being used.

If you're running a template from the command line, add -v for "verbose" (i.e. "wordy",) like so:

```
forge -v [tid]
```

or:

```
forge [tid] -v
```

If the data you get back isn't quite enough, try upping the level of wordiness:

```
forge -v2 [tid]
```

Levels 1 through 9 are available. Level 1 is what you get if you don't specify a number.

Exercises:

Ex. 1: Modify your template from 'More file-ability,' Ex. 1. Add tags to print your variables and data tags to the command line. Run the template from the command line. Verify that the extra data does not appear in the file Forge output. Run the template from the web. Does the extra data show up there?

Settings

You could type in the full root for every link in your template, but that's a lot of typing - and most of it will be redundant. So use <mtl settings>:

```
<mtl settings root="[domain]:/web/[domain]/htdocs">
```

will set the root for all subsequent <mtl a>'s in the template (so you probably want it at the top.)

Other settings (bold = default setting):

```
case_sensitive = on | off
compress = on | off
debug = on | off
default_host= hostname
file_locking = on | off
missing_links = on | off
multi_category= on | off
tp = template_property
root =
```

- CASE_SENSISTIVE is used to make the IF statement treat upper-case and lower-case characters different when comparing text. Normally the statement <MTL IF {'a' = 'A'}> will be true, but if a SETTINGS tag is used to specify CASE_SENSITIVE=ON, all subsequent statements like this will be false.

- COMPRESS is used to minimize the size of HTML output from a template. COMPRESS=ON will cause every run of whitespace characters (space, tab, return) with a single space. Be aware that this will usually put the entire html page on one line - and some browsers don't like that.

- DEFAULT_HOST is used mostly when you need to move a template from the file system to the Template Manager database. Using DEFAULT_HOST you can specify which server and directory path the template came from so that subsequent file references will work correctly. (Example: DEFAULT_HOST = augusta.com:/web/augusta.com/htdocs/my_dir). Any time a file is referenced using a relative file path (one that doesn't begin with a slash or hostname) the DEFAULT_HOST value is prepended to that file name to locate the file. The default value for DEFAULT_HOST is your current work directory.

- FILE_LOCKING is used to lock output files while they are being written to so that no one else can read the file until the write is complete. Normally file locking is not needed and will only succeed if the person running the template is also the owner of the file being locked.

- MISSING_LINKS - When an MTL A tag is used and the file specified by the HREF is not found, the link to that file is not included in the HTML output. Later if that file is created from a template, the previously output HTML file will be edited automatically to insert the link. In some cases this process can take a few seconds and if not needed MISSING_LINKS=OFF can be used to improve performance.

- MULTI_CATEGORY - Specifies whether to allow multiple categories to be associated with a single ad in the <mtl classifieds.ad> query tag. By default, only one category is associated with a single ad. However, if you have set up ads which use multiple categories per ad and you wish to pull these ads back based on any category that the ad is a member of, you will need to set this setting to ON.

- TP explicitly sets the Template Property. The Template Property determines which Property's templates are searched in the Template Manager database when a template is needed and is used mostly by the <MTL INCLUDE NAME=> tag. For dynamic templates, the Template Property normally is set automatically from the host name used in the URL. If a template uses INCLUDE NAME= and is run from the UNIX command line, the Template Property needs to be explicitly specified either on the command line, in a SETTINGS tag or in the INCLUDE tag itself.

Web tags

You'll occasionally need to know about how someone got to your template. Examples in this chapter will refer to this url:

```
http://boo.hoo.com/forge?category=test&temp_type=detail&color=yellow&size
=small
```

The following tags deal directly, and *only* with the URL used to access the template they're contained in - and the referrer the user got to the template from. They are available outside of any query. Note that they are exceptions to the rule - only one dot, but they're all **data tags**:

- `<mtl web.referring_host_name>` - the domain the user came to the current template from
- `<mtl web.host_name>` - the domain used to get to the template. Would return boo.hoo.com from sample URL.
- `<mtl web.path>` - the URL from the host name to the question mark. Would return /forge from sample URL.
- `<mtl web.query_string>` - everything after the question mark. Would return category=test&temp_type=detail&color=yellow&size=small from sample URL.
- `<mtl web.url>` - the path and query string (or path and key/value pairs from a method=post form submission) used to access the template, i.e. everything after the domain name. Would return /forge?category=test&temp_type=detail&color=yellow&size=small from sample URL.
- `<mtl web.absolute_url>` - the full URL used to access the current template.

The difference between `web.url` and `web.absolute_url` is the difference between relative and absolute linking. If you're using these tags to generate navigation links among a set of templates, the use of `web.url` is both recommended and much preferred. Why? Because using web.absolute_url in these situations can interfere with page caching.

There are two special modifiers available to the tags that have the query string in what they return, i.e. `web.query_string`, `web.url` and `web.absolute_url`:

- **`remove_arg=`** this modifier takes the name of a URL key. Use it to remove the specified key and its value from the URL. For instance, this code:

  ```
  <mtl web.url remove_arg=color>
  ```

  Would give this output:

  /forge?category=test&temp_type=detail&size=small

To remove more than one key/value pair, use multiple remove_arg= modifiers:

```
<mtl web.absolute_url remove_arg=color remove_arg=size>
```

Returns:

http://[domain]/forge?category=test&temp_type=detail

- **arg=** this modifier takes a `key=value` pair (in that format) and changes the value of the specified key to the specified value. If the specified key is not in the url or has no value, it will be added. To change the values for multiple keys, use the modifier multiple times. This code:

  ```
  <mtl web.query_string arg="color=blue">
  ```

  Would give this output:

  category=test&temp_type=detail&color=blue&size=small

If your user authenticated (logged in) to get to your templates, these two pieces of data will be available on every page password-protected page:

- `<mtl web.user_name>` - the login the user used
- `<mtl web.user_id>` - the user's unique user id. This corresponds to `<mtl reg.user.id>`.

You can use a `<mtl weburl.arguments>` query to run through the key/value pairs in your URL:

```
<mtl weburl.arguments>
     key:  <mtl weburl.arguments.key> |
     value: <mtl weburl.arguments.value><br>
</mtl weburl.arguments>
```

From the sample URL, the above code returns this:

```
key: cateogory | value: test
key: temp_type | value: detail
key: color | value: yellow
key: size | value: small
```

This query has two **qualifiers**:

- `key=` the name(s) of the key or keys you're looking for. Realize, tho, that if you already know the name of the key, you could just skip this query and use `<mtl $[key name]>` instead.

- `key_startswith=` use a single letter or series of letters here. For instance, you could use this to find all the keys that start with "a" or that start with "cat." You cannot, however, use this qualifier to find both. Specifically, `key_startswith=cat,a` won't work. Nor could you use this qualifier twice to find both. The query will use only the last `key_startswith=` in the query.

Exercises:

Ex. 1: Write a template that prints to the screen a list of the key/value pairs from the URL, and a link back to itself.

Include at least 1 key/value pair in the URL that isn't needed for navigation to the template. The value should have spaces or other non-alpha characters. Make sure your link works.

Cookies

While you can pass a great deal of information back and forth between templates in URL's, there are times when that ability just isn't enough. In those cases, cookies might be what you need.

MTL has two cookie tags:

- `web.cookie(cookie name)` - Use this tag to get the value of the specified cookie. Like any other data tag, you can output the returned value, store it in a variable or use it in a calculation.
- `web.set_cookie` - Use this tag to set a cookie. It must come before any other output from the template (this includes blank lines.) Here are the modifiers this tag takes:

  `name=` the name of the cookie. This is **required** (not by MTL, but by the cookie spec.)

  `value=` the value to store. This is **required** (not by MTL, but by the cookie spec.)

  `domain=` which domains can read the cookie. Defaults to the domain it was set from. Set this if you'd like to be able to read your cookies from other parts of your site.

  For example, let's say your domain is morris.com, and you're setting a cookie from stats.morris.com, but you'd like to be able to read it from calendar.morris.com. You'd set this value to "**.**morris.com" - note the leading dot.

  `path=` sets the path the cookie is valid within. Defaults to the path the cookie was set from, generally "/forge" in our case.

  `secure=` use YES or NO here. This indicates whether to restrict transmission of the value to SSL situations. The default, which is what you'll usually want, is NO.

  `expire=` the date and time the cookie should expire on. Leave this out and you get the default: the cookie expires at the end of the session (what this actually means varies browser to browser.) Values should be formatted like this: "Wed, 14 Oct 1997 06:41:40 GMT". If you're using a MTL-generated date, here's how it might look:

  ```
  EXPIRE="{time.date(today+8) fmt='Dy, dd Mon YYYY 06:41:40 GMT'}"
  ```

  Notice, I've filled in the time by hand since `time.date` doesn't return a time.

Example:

Here's a corny, but quick example:

```
<mtl $cookietest={web.cookie(test)}>
<mtl web.set_cookie name="test" EXPIRE="{time.date(today+365) fmt='Dy, dd
Mon YYYY  06:41:40 GMT'}" value="{time.date}">

<mtl if {$cookietest!=""}>
     Welcome back.
     We've missed you since your last visit on
     <mtl time.date($cookietest) fmt="Dy., Mon ddth"><p>
</mtl if>
```

Exercises:

Ex. 1: Write two templates; one to set a cookie and the other to pick the value back up and display it.

Cobranding

If you've got cobranding implemented on your static site and would like to extend it to your dynamic content, you'll use the URL tags and the ability to include static files to accomplish that.

The template:

There are three steps. First you'll need a cobrand template. This one is lifted from ehorseads.com. For brevity, explanations are in-line:

```
<!mtl "if there's no $cobrand value but there is a referrer">
<mtl if {$cobrand='' and web.referring_host_name!=''}>
     <!mtl "and if the referrer isn't the current (database) hostname">
     <mtl if {web.host_name!=web.referring_host_name}>
          <!mtl "set $cobrand to referrer">
          <mtl $cobrand={web.referring_host_name}>
     </mtl if>
</mtl if>

<!mtl "if $cobrand is still blank, default it back to the main cobrand">
<mtl if {$cobrand=''}>
     <mtl $cobrand="horsecity.com">
</mtl if>

<!mtl "Reset out hostnames we don't want to the default cobrand">
<mtl if {$cobrand="classifieds" or $cobrand="chat.horsecity.com"}>
     <mtl $cobrand="horsecity.com">
</mtl if>

<!mtl "we'll be using this value to include files
     & file names are case-sensitive, so reset case">
<mtl $cobrand="{$cobrand case=lower}">

<!mtl "strip 'www.' from cobrand if it's there">
<mtl if {STRING.CONTAINS($cobrand,"www.")>0}>
     <mtl $start={string.length(www.) +1}>
     <mtl $cobrand={STRING.SUBSTR($cobrand,$start)}>
</mtl if>
```

Copy the above code, modify it to match your web address(es) and/or needs and save it into an include template named "cobrand." If you're going to cobrand content from multiple categories (calendar, classifieds &etc) create a new category named "cobrand" and save the template in there. Otherwise, save it into the single category whose content you'll be cobranding.

Using your cobrand template

Now, include your cobrand template into all the templates whose content you want cobranded. (Don't worry, your include templates and the templates you're including them into don't have to be in the same category.) You're half done.

Now, for the content you want cobranded, you'll insert include calls like this:

```
<mtl include
file="[domain]:/web/[domain]/htdocs/ssi/{$cobrand}/[file_you_want_include
d]}">
```

Note the curly braces around `$cobrand`; you're already in an mtl tag.

Now pass your cobrand value forward, by adding one of these two versions to all your templates' links to dynamic content:

```
&cobrand=<mtl $cobrand>
&cobrand={$cobrand}
```

What happens:

- When you include your cobrand template, you make sure your cobrand variable is "populated" with a value, and cleaned up. (You need to do the "case=lower" step each time through.) If it's your first time into your templates, i.e. you're coming from a static page, $cobrand will be populated with the referring hostname, i.e. "[cobrand].[yoursite.com]." Otherwise, you'll continue with what was passed forward from the last template-generated page.
- Then when you include your static cobrand files, those include paths will be based on the value of cobrand, so you'll get the right files in the right places.

Be aware that if you're using the same static cobrand includes to cobrand both static and dynamic content, you'll need to make sure URL's and image calls in those includes are full, rather than relative.

Universal qualifiers

Just as there are a few universal data tags (like index) there are a few universal query qualifiers. Almost without exception, these qualifiers can be used in any query:

`No_data=` tells Forge what to do if it doesn't find any records that match your query. There are 3 options:

- `No_data=abort` tells Forge that if your query matches 0 records, it should halt output. So if you're running a template from the command line and one of the queries in your template returns zero results, '`no_data=abort`' tells Forge not to output the file at all - even if other queries in it did return results.
- Abort is the default behavior. So leaving `no_data=[x]` out of your query is the same as specifying `no_data=abort`.
- `No_data=ok` tells Forge to skip the query if no records match it & keep going anyway. So even if one of your queries did return zero records, Forge will output your file(s).

  When you're accessing a template from the web, `no_data=abort` looks, works and feels just like `no_data=ok`. Why? Because "abort" only aborts the output of files (which doesn't happen anyway when you call a template from the web.) It doesn't abort the processing of the template.
- `No_data=continue` tells Forge to act as if one record with blank values were found. In this one instance, when `no_data=continue` is part of your open query tag and Forge finds zero records that matched, index will be 0. Which can be very useful.

  It's the only time this test will evaluate to true:

```
<mtl if {category.table.index=0}>
      Sorry, no records match your search.<p>
</mtl if>
```

`Blanks=` tells Forge what to drop in if one of the data tags returns a blank. I.e. `blanks=" "` in your query would tell Forge to insert a non-breaking space for any data tag that returned a null value.

This can be over-ridden on a tag-by-tag basis:

```
<mtl mdw.personnel blanks=" - ">
          <mtl mdw.personnel.name>
          <mtl mdw.personnel.eye blanks="tiger">
          <mtl mdw.personnel.department><br>
</mtl mdw.personnel>
```

Might return something like:

```
Jen
  tiger
  support<br>
David
  blue
   - <br>
Amy
  tiger
   - <br>
…
```

David and Amy's department fields were blank & those blanks got filled by the string specified in the query. But Jen and Amy's blank eye fields were filled with the string specified locally for that tag.

Pagination

Dividing results into pages

There are two more universal query qualifiers: `records_per_page=` and `page=`. These two qualifiers will allow you to paginate (or break into pages) the results of your query.

So if your query brings back 400 results, but you want those results presented in digestible chunks of 10, you do this:

```
<mtl category.table … page=$page records_per_page=10>

</mtl category.table>
```

Which limits the first page of results to the first 10 records that match the query.

But how do you get to the next set of 10? Pagination reporting and navigation are handled with these **data tags**:

- `category.table.page_number` - the number of the current page.
- `category.table.page_count` - the number of pages found. This calculation is the total number of results (`category.table.num_rows`) divided by the `records_per_page=` value.
- `category.table.next_page` - the full URL to the next page. Add `text="[link text]"` to it:

  ```
  <mtl category.table.next_page text="Next!">
  ```

  and it'll produce the entire link string for you if there is a next page, and suppress the link if there's not:

  ```
  <a href=[url]>Next!</a>
  ```

  Otherwise you'll need to wrap <a href=""> around it:

  ```
  <a href="<category.table.next_page>">Next!</a>
  ```

- `category.table.prev_page` - the URL to the previous page. Similar to next_page, you can add `text="[link text]"` and have the entire link string produced for you.

The next_page and prev_page tags link back to the template they're generated from, and add "`page=[page number]`" to the URL. That page number is picked up by the "`page=$page`" qualifier in the query to display the correct "chunk" of records.

That means that for pagination to work you have to have **both qualifiers** in your query. Leave off records_per_page, and all the matches show up on the first page. Leave off `page=$page` and you'll never be able to get to the second page of results.

There's one more pagination data tag to consider:

```
category.table.page_nav_bar
```

It outputs a list of linked page numbers like so: "1 2 3 ... 9 10 11 ... 18 19 20." Every page number but the current one will be linked. Like `next_page` and `prev_page`, these links also add page=[page number] to the URL.

There are a number of modifiers that apply specifically (and only) to page_nav_bar:

- `min_first=` The number of page numbers to display at beginning of list (defaults to 3.)
- `min_last=` The number of page numbers to display at end of list (defaults to 3.)
- `range=` The number of page numbers to display on either side of the current page (defaults to 1.)
- `delimiter=` Delimiter to use between page numbers. The default is a space: " ".
- `ellipsis=` Replacement for the default ellipsis: "…".
- `color=` Font color for page numbers

Here's a pagination example*:

```
<mtl biz.business market=Augusta records_per_page=10 page=$page
orderby=name>
     <mtl if {biz.business.index=1}>
          Page <mtl biz.business.page_number>
          of <mtl biz.business.page_count> pages found<p>
          <mtl biz.business.prev_page text="Last">
             

          <mtl biz.business.page_nav_bar color="#cc0000">

             
          <a href="<mtl biz.business.next_page text="Next">
          </center>
     </mtl if>
     <mtl biz.business.name><br>
     <mtl biz.business.street1><br>
     <mtl biz.business.city><p>
</mtl biz.business>
```

This produces pagination that the browser renders like this:

Page 13 of 47 pages found

Last 1 2 3 ... 12 13 14 ... 45 46 47 Next

[actual business listings omitted]

Note that in the page range, "13" isn't linked. That's because we're on page 13. The red font color got applied to the linked page numbers generated by `page_nav_bar`, but not to "Last" and "Next." Those words and links weren't generated by `page_nav_bar`. We've got the default number of pages at the beginning and end of the range (1-3 and 45-17) and the default number of pages (1) on either side of the current page.

This version:

```
<mtl biz.business market=Augusta records_per_page=10 page=$page
orderby=name>
     <mtl if {biz.business.index=1}>
          <mtl biz.business.page_nav_bar range=2 delimiter=" | "><p>
     </mtl if>
</mtl biz.business>
```

Gets us:

1 | 2 | 3 | ... | 11 | 12 | 13 | 14 | 15 | ... | 45 | 46 | 47

Pagination, Random & Rseed

So now you've got your biz.business listings from Augusta broken up into pages.

But what happens if you used `orderby=random` in your template?

```
<mtl biz.business market=Augusta records_per_page=10 page=$page
orderby=random>
```

For instance, let's say you're dealing with premium business listings. Everyone who shows up here is someone who paid you big bucks. But if you use `orderby=name` in your template, Zeb's Bait Shoppe, will always show up dead last – even though Zeb paid just as much as Aaron did to have Aaron's Crickets & Worms listed. And Zeb's not happy about that.

If you use `orderby=random`, on the other hand, Zeb has the same chance as Aaron of showing up on the first page. Now everyone's happy.

Except…

Except if the output is random, what keeps the businesses that randomly showed up on page 1 from randomly showing up again on page 2? Absolutely nothing. And going back to page 1 from another page of results is also a problem. Since the results are randomly generated, your chances of getting the same (random) page 1 as the first time are pretty slim.

That's where `rseed` comes in.

The ugly truth of the matter is that random, when generated by a computer, really isn't. It's random enough to fool the human eye, but it's not *really* random. Computers generate random number sequences using equations - equations that start from a base number, called a seed.

Because computers generate "random" from equations, if you seed the equation with the same number over and over again, you'll get the same "random" sequence over and over again.

This isn't something you have to do manually, though. If you use the pagination data tags described above, Forge will take care of it for you.

If you're paginating `orderby=random` results you'll notice that `&rseed=[a number]` is added to the end of the URL's the pagination tags produce. This passes forward (and backward) the number the equation was seeded with the first time. So on page 2 of your randomly ordered result set you don't get any of the records that showed up on page 1. And if you return to page 1, you'll get the same set of results you did the first time.

Exercises:

Ex. 1: Start with the biz.business pagination example above and modify the pagination so you have a link to the first page, the last page, and links to the page before and the page after the current page.

* Note these examples use biz.business tags to demonstrate pagination, but do not document biz.business tags. It's not the purpose of this section of documentation to cover the biz.business tags. Biz.business tags will be covered in depth in another section.

Skipping

The best templates are written so they hone directly in to the set of results you want - you don't have to filter out a lot of irrelevant results, - either on the screen or in your template (using if statements.)

That said, there are still times when even the best template will bring back more than the results you wanted. Sometimes it's that you only wanted the first 20 results; other times it's that your result set includes multiple records with only minor differences among them.

For example, let's say you wanted to show a list of restaurants in a given town. And in addition to the non-chain local restaurants, you get "Burger Queen" 5 times and "McDougals" 8 times and 12 "Waffle Huts" & etc. But you only want the chain restaurants to show up once per chain.

The hard way to deal with this involves setting variables & using if statements.

The easy way is to qualify your **close** query tag:

```
<mtl biz.business market=Augusta business_type=dining orderby=name>
     <mtl biz.business.name><br>
</mtl biz.business next=name>
```

Putting `next=[colname]` in your close query tag tells Forge to skip to the next record with something different in the specified column. So, `next=name` tells Forge to compare the name value of the current record with the name value of the next record. If they're the same, skip it & look at the name value of the following record. Forge keeps skipping until it comes to a record with a different value in the name field - and outputs that one.

Note that `next=[colname]` only works in the close query tag. It does not work in the open query tag. If you put `next=[colname]` in the open query tag, Forge won't know what you're talking about. So only put it in the close query tag.

The other skip qualifier is `max=`. If you only want the first, say 20, results from your query, you can suppress the rest of them with if statements, or you can do it the easy way:

```
<mtl biz.business market=Augusta business_type=dining orderby=name>
     <mtl biz.business.name><br>
</mtl biz.business max=20>
```

Adding max=20 to the close query tag tells Forge to spit out the first 20 results and skip the rest. As with `next=`, `max=` works only in the close query tag. It does not work in the open query tag.

You can use both skip qualifiers at the same time if you like:

```
<mtl biz.business market=Augusta business_type=dining orderby=name>
     <mtl biz.business.name><br>
</mtl biz.business next=name max=20>
```

Gives you the first 20 uniquely named restaurants in Augusta.

But before you use either skip qualifier, make sure you've made your query as specific as possible. Don't use a skip qualifier in your close query tag to narrow down your results when you could get the same narrowed set by adding another qualifier to your open query tag.

Why? Qualifiers in the open query tag influence how the records are "picked." Qualifiers in the close query tag influence how they're output.

Let's say you write a query that returns 200 results. When you run that template, it takes Forge time to gather those results. Once all the results are in, Forge can sort them according to your orderby qualifier. Only when all 200 results are gathered and sorted does Forge begin spitting them out. Let's say that takes 30 seconds. That's a 30 second wait before Forge even begins sending data to the browser.

If you then add a skip qualifier to your close query tag (because you didn't actually want all 200 results,) and run the template again, it's going to take Forge that same 30 seconds to gather and sort those same 200 results before Forge outputs the small fraction of them you narrow it down to with your skip qualifier.

But if you can get the same "narrowed" set of results by adding another qualifier to your open query (i.e. if you give Forge better criteria for picking the records that "match") then Forge spends less time gathering and sorting the records and starts sending output to the browser much more quickly.

Exercises:

Ex. 1: `Next=[colname]` works best when the column you're "nexting" on is the same one you're ordering by. Why?

The MOST IMPORTANT tag of all

Every language has comment syntax and MTL is no exception.

But why is the comment the most important MTL tag of all?

Because eventually the templates you're writing will break. Or you'll need to modify them. Or add functionality. Or take it away. Or worse, someone will come to you in total awe of what you did and ask you to explain it.

And when you go back to those templates - next week, next month, two years from now - you'll have no idea what you did … Unless you use comments as you write your templates.

Comments are the road map you leave yourself (or the people who'll have to come after you) to understanding your template. No matter how clear-cut, how obvious you think what you're writing is today, I can promise that it won't be later.

Each coder has his/her own style, but types of comments to consider are:

- Of *course* you've named your templates to indicate what they're supposed to do. But if it's not completely clear, consider adding comments to that effect.
- Of *course* you've named your variables to indicate what's in them (or what's supposed to be in them,) but if what they're for is not blazingly obvious, you might add comments to indicate it.
- Segment comments - to delineate the start and end of sections of a template.
- Logic/reasoning comments - consider adding comments to complicated if statements to indicate what they're supposed to be "true" for. Complex query tags, or sets of query tags are another good candidate.
- Comment the flow and structure of your template - what you're *supposed* to be accomplishing, how you're trying to do it and why.
- This is by no means a comprehensive list, and it's much easier to use too few comments than to many - so don't restrict yourself to only the kinds of comments on this list.

`<!mtl` declares the start of a comment. If it's immediately followed by a quote mark (double or single) everything until close quote, close bracket (`">` or `'>`) is commented out. Without quotes, a comment extends to the first close bracket (>).

To comment a single MTL tag, add an exclamation point (`!`) between the open brace (<) and the "`mtl`":

```
<!mtl "--- Current TimeStamp ---">
Time: <!mtl "time.hour>:<mtl time.minute>.
```

Returns:

Time: :14.

And this:

```
<!mtl "--- Current TimeStamp ---">
Time: <!mtl time.hour>:<!mtl time.minute>.
```

Returns:

Time: :.

Note that commenting the two time tags doesn't suppress the non-MTL text between them. To comment more than just a single tag, add quotes, like so:

```
<!mtl "--- Current TimeStamp ---">
Time: <!mtl "time.hour>:<mtl time.minute">.
```

Returns:

Time:.

As you've seen in the code above, it can be helpful to use a different style in the comments intended to communicate, versus the comments you use to suppress MTL.

The difference between an html comment and a MTL comment is that Forge not only ignores the contents of a MTL comment - it also suppresses the contents, while html contents are output like normal text. If you're trying to keep your page size low, this can be a crucial difference:

```
<mtl loop times=5>
     <!-- loop index -->
     line <mtl loop.index><br>
     <!mtl loop index>
</mtl loop>
```

Here's the browser source that returns:

```
<!-- loop index -->
line 1<br>
<!-- loop index -->
line 2<br>
<!-- loop index -->
line 3<br>
<!-- loop index -->
line 4<br>
<!-- loop index -->
line 5<br>
```

In this example, the html comments would be more than half the file size.

Reverse Publishing Note:

- You can suppress an unwanted line break by putting a MTL comment at the end of the line it's coming from.

Exercises:

Ex. 1: Add comment tags to all your templates.

Exercise Answers:

Accessing your templates

Ex. 1: For help with this, contact your Support rep.

Testing Time

Ex. 1:

```
<mtl time.string(tomorrow)>
<mtl time.string(next_Thursday)>
<mtl time.string(today+48)>
```

Ex. 2:

```
<mtl time.string(11/12/2030)>
```

Ex. 3:

```
<mtl loop times={2030-time.year}>
     <mtl $date=11/12/{time.year+loop.index-1}>
     <mtl $date date_fmt="YYYY: Day"><br>
</mtl loop>
```

Ex. 4:

```
<mtl $bday=11/12/{time.year}>
<mtl loop times=time.days_in_month($bday)>
     <mtl $date=11/{loop.index}/{time.year}>
     <mtl if date=$date eq=$bday>
          <b><mtl $date date_fmt="Day, Mon. dd"></b><br>
     <mtl else>
          <mtl $date date_fmt="Day, Mon. dd"><br>
     </mtl if>
</mtl loop>
```

Variables from 'the outside'

Ex. 1: Template copied from Ex. 2 above. New code in italics

```
<mtl time.string(11/12/{$year})>
```

Ex. 2: Template copied from Ex. 1. New code in italics.

```
<mtl if {$year=''}>
     <mtl $year=2030>
</mtl if>
```

```
<mtl time.string(11/12/{$year})>
```

Forms

Ex. 1:

```
<form action="http://[your listener]/forge" method=get>
     <input type=hidden name=category value=[your category]>
     <input type=hidden name=temp_type value=[your temp_type]>
     <input type=hidden name=tl value=[your tl]>
     Please enter a 4-digit year:
     <input type=text name=year size=4>
     <input type=submit value="Get the date">
</form>
```

Email

Ex. 1: Consult examples in chapter

Files

Ex. 1: Template copied from Ex. 4 above. New code in italics.

```
<mtl output file="[domain]:/web/[domain]/htdocs/test/file1.txt">
<mtl loop times={2030-time.year}>
     <mtl time.string(11/12/{time.year+loop.index}) fmt="YYYY: Day">
</mtl loop>
<mtl a root="[domain]:/web/[domain]/htdocs/"
href="test/file2.txt">Test2</mtl a>
</mtl output>
```

Ex. 2:

```
<mtl output file="[domain]:/web/[domain]/htdocs/test/file2.txt">
     boo!
</mtl output>
```

Ex. 3:

```
<mtl include file="[domain]:/web/[domain]/htdocs/test/file1.txt">
<mtl include name="hello world" tp="[your tp]"
```

Debug & Print

Ex. 1: Template copied from Ex. 1 above. New code in italics.

```
<mtl output file="[domain]:/web/[domain]/htdocs/test/file1.txt">
<mtl loop times={2030-time.year-1}>
     <mtl time.string(11/12/{time.year+loop.index-1}) fmt="YYYY: Day">
<mtl print year={time.year+loop.index-1}>
```

```
</mtl loop>
<mtl a root="[domain]:/web/[domain]/htdocs/"
href="test/file2.txt">Test2</mtl a>
</mtl output>
```

Web tags

Ex. 1: If you're using I.E. or Netscape 6 to view the output of this template, you could get away with omitting "fmt=url" in the weburl.arguments.value tag, but the resulting URL would break for users of Netscape 4.* and below.

The MTL below accommodates the broadest possible audience.

```
<mtl weburl.arguments>
     key:  <mtl weburl.arguments.key><br>
     value: <mtl weburl.arguments.value><p>
     <mtl
$queryString="{$queryString}&{weburl.arguments.key}={weburl.arguments.val
ue fmt=url}"><!mtl "no line breaks!">
</mtl weburl.arguments>

<a href="http://<mtl web.host_name>/forge?<mtl $queryString>">Click here
to refresh this page</a>
```

Cookies

Ex. 1:

Template 1: Make sure this is the first line of output. You can have calculations, but blank lines above this

```
<mtl web.set_cookie name="test2" EXPIRE="{time.date(today+365) fmt='Dy,
dd Mon YYYY  06:41:40 GMT'}" value="{time.date}">

Hi
```

Template 2:

```
<mtl $cookietest={web.cookie(test)}>
<mtl if {$cookietest!=""}>
     Looks like you were at the first template on
     <mtl time.date($cookietest) fmt="Dy., Mon ddth"><p>
</mtl if>
```

Pagination

Ex. 1: Actual data output omitted.

```
<mtl biz.business market=Augusta records_per_page=10 page=$page
orderby=name>
     <mtl if val=biz.business.index eq=1>
          <center>
```

```
          Page <mtl biz.business.page_number> of <mtl
biz.business.page_count> pages found<p>

          <mtl biz.business.page_nav_bar range=1 min_first=1 min_last=1>
          </center>
     </mtl if>
</mtl biz.business>
```

Skipping

Ex. 1: Left to the reader.