Table of Contents

What is MTL (Metal)

A brief description of MTL

MTL, pronounced "metal" stands for "Morris Templating Language."

It was written to give non-programmers tools to pull data out of Morris Digital Works' (Oracle) database.

MTL is a proprietary Morris tool used throughout the Morris chain, and increasingly throughout the newspaper industry.

The very basics

MTL is a Query

The most important part of MTL is the query tag.

Look up "query," and you'll learn it's a question or request for information.

In non-MTL terms, we can look at a query like this:

I ask Morris Digital Works' Office Manager, Barbara Slack, for a list of blue-eyed MDW employees. She gives me this list: Kevyn, Barbara, David, Chris, Wayne, Brent.

I've queried her and gotten back a list of results.

That, in a nutshell, is all MTL does: it lets you [1)] ask questions, and [2)] deal with the results.

Getting warmer

Example Table Structure

For practical purposes, you can imagine the database as a pile of spread sheets (engineers call them "tables.") To keep them organized, they're grouped into categories (think of file folders) like classifieds, sports and calendar. Within those categories, each spreadsheet, or table, has a name.

Here's the personnel table in the MDW category:

name	eye	hair	department
Kevyn	blue	brown	support
Jerome	brown	salt & pepper	support
Natasha	brown	black	support
Barbara	blue	blond	executive
David	blue	red	NOC
Michael	brown	salt & pepper	executive
Felecia	green	brown	content
Toby	green	blue	content
Chris	blue	brown	engineering
Wayne	blue	brown	engineering
Nik	brown	black	content
Angelia	brown	blond	content
Connie	brown	black	sales
Brent	blue	brown	NOC
Bill	grey	none	sales

Going back to the personnel example, let's say we ask Barbara again for that list of blue-eyed employees.

So she looks in the MDW folder (category) of spreadsheets (she has other folders) and finds the one named personnel. Then she goes down the column labeled "eye" and finds six matches on the word "blue."

name	eye	hair	department
Kevyn	blue	brown	support
Jerome	brown	salt & pepper	support
Natasha	brown	black	support
Barbara	blue	blond	executive
David	blue	red	NOC
Michael	brown	salt & pepper	executive
Felecia	green	brown	content
Toby	green	blue	content
Chris	blue	brown	engineering
Wayne	blue	brown	engineering
Nik	brown	black	content
Angelia	brown	blond	content
Connie	brown	black	sales
Brent	blue	brown	NOC
Bill	grey	none	sales

For each of the rows where she finds a match, she looks in the name column and gives me what's there.

So the list I get back will be: Kevyn, Barbara, David, Chris, Wayne, Brent.

So, what's this 'Forge' thing?

What Forge does

In our first examples, we talked about asking Barbara to give us a list of people who matched certain criteria.

We asked her for data, and told her how we wanted the results handed back ("names only," we said.)

But we're about to talk about putting our examples in MTL, and I'll start talking about sending the requests to Forge.

Forge is the program that reads our templates and carries out their instructions.

When we write a MTL query, it's Forge that reads it, fetches the right data from the database and spits it out in the format we asked for.

Almost there

Query and Data Tag Syntax

Every MTL tag starts with "`<mtl  `" - which I call the "Hey, Forge!" part of the tag, and ends with ">".

Once we've got Forge's attention, we have to tell it where to go. So after "`<mtl  `" we specify which category to go to and then which table we're dealing with: "`<mtl category.table>`"

There are two basic types of MTL tags: query tags and data tags.

A query tag is where we ask the question (Which MDW employees have blue eyes?)

In a query we tell Forge which category to go to and which table in the category to look at (MDW employees.) If we don't want every record in the table, we need to tell Forge how to decide which rows of the table to return. We do that by specifying columns, and the value(s) that should be in those columns in the records we're looking for (i.e. look in the "eye" column for "blue".) We do that with this type of syntax: `<mtl category.table colname=value>.`

With data tags we specify the pieces of information we want from the records that are returned by the query (Give me what's in the "name" column) with this type of syntax: `<mtl category.table.colname>`.

Pseudo MTL

Basic Rules of MTL

So let's translate our personnel example to pseudo-MTL* and pull out the name, and department of each record that matches our query:

```
<mtl mdw.personnel eye=blue>          open query tag
     <mtl mdw.personnel.name>               data tag
          <mtl mdw.personnel.department><p> data tag
</mtl mdw.personnel>                  close query tag
```

The output from this would be:

```
Kevyn
        support<p>
Barbara
        executive<p>
David
        NOC<p>
Chris
        engineering<p>
Wayne
        engineering<p>
Brent
        NOC<p>
```

A couple of things to note about this example:

- Your output will be formatted like your MTL, where your MTL has spaces or line breaks your output will have spaces or line breaks.
- The open and close query tags have only one dot (.) in them (category.table).
- There are no colname=value pairs in the close query tag.
- The data tags have two dots (category.table.colname).
- The "category.table" parts of the open and close query tags (mdw.personnel) must match each other.
- The "category.table" part of the data tags must match the query tags.
- We can access the data from any column, not just the ones we modified the query by.
- Forge only "pays attention to" or interprets tags that follow this format: <mtl [something here]> (or </mtl [something here]>). Everything else gets output just as it appears - including spaces. Generally Forge will repeat in the output the spacing in the template. That's why we get an extra return and tab before each department. Of course, in HTML output this won't usually matter, but it's something to be aware of.

*there's not actually an `mdw.personnel` table in the MDW database

Almost there (part 2)

Multiple Query Qualifiers

In our personnel example above, we've only modified the query by one column name; `eye=blue`. But we can modify by as many columns as we want.

For instance, this query:

```
<mtl mdw.personnel eye=blue hair=brown>
```

name	eye	hair	department
Kevyn	blue	brown	support
Jerome	brown	salt & pepper	support
Natasha	brown	black	support
Barbara	blue	blond	executive
David	blue	red	NOC
Michael	brown	salt & pepper	executive
Felecia	green	brown	content
Toby	green	blue	content
Chris	blue	brown	engineering
Wayne	blue	brown	engineering
Nik	brown	black	content
Angelia	brown	blond	content
Connie	brown	black	sales
Brent	blue	brown	NOC
Bill	grey	none	sales

would give us a list of blue-eyed people with brown hair: Kevyn, Chris, Wayne, Brent.

On the other hand, this query:

```
<mtl mdw.personnel>
```

Would return a list of everyone in the table: Kevyn, Jerome, Natasha, Barbara, David, Michael, Felicia, Toby, Chris, Wayne, Nik, Angelia, Connie, Brent, Bill.

General Format

Query and Data Tags

The general format for a MTL query and data tag set follows the format we've seen above:

```
<mtl category.table colname=value colname=value … >
     <mtl category.table.colname>
     <mtl category.table.colname>
     …
</mtl category.table>
```

Generally you can modify a query by any column in the table.

Generally you can output the data in any column.

And as long as you follow this format, you can pull any data you want from any table in the database.

So the question becomes: How do I know what the categories and tables and columns are named?

And the answer is : consult the MTL Tag Quick Reference.

The first page is a list of categories. Click through on the category to get its table list. Click through on a table name, and you'll get a list of query modifiers and data tags for the table.

And basically, that's it. If you're with the game to this point, you're ready to go forth and write templates.

But wait, there's more!

How the Query Structure Works

The normal flow, or execution of a template is from top to bottom, like a browser executes html, like you read a page, like water (to wax poetic) flows to the sea.

"But wait," you're thinking, "in the personnel example, you only typed the '<p>' tag once but it came out for each record the query brought back."

That *is* what you're thinking, right?

Okay, let's revisit that example. Let's say our template is this:

```
These MDW employees have blue eyes:<P>
     <mtl mdw.personnel eye=blue>
     <mtl mdw.personnel.name>
          <mtl mdw.personnel.department><br>
</mtl mdw.personnel>
```

And Forge hands us back this:

```
These MDW employees have blue eyes:<P>
Kevyn
  support<br>
Barbara
  executive<br>
David
  NOC<br>
Chris
  engineering<br>
Wayne
  engineering<br>
Brent
  NOC<br>
```

So why do we get "These MDW employees..." only once, but the
 for each record?

Forge normally executes templates from top to bottom … unless we tell it not to. The
, like the data tags for name and department, is inside a **control structure** - a special instruction *not* to go straight from top to bottom.

This template (notice, there's no close query tag):

```
These MDW folks have blue eyes:<P>                text
<mtl mdw.personnel eye=blue>                      open query tag
     <mtl mdw.personnel.name>                     data tag
          <mtl mdw.personnel.department><br>      data tag
```

Would give us this output:

```
These MDW folks have blue eyes:<P>
Kevyn
  support<br>
```

Does that mean Forge only found one record for the query this time? No. It still found all 6 people. But without the close query tag, Forge runs through the template like water to the sea: straight from top to bottom without stopping. And only the data for the first record gets spit out because Forge didn't know what to do with (had no instructions for) the other records it found.

By pairing the open query tag with a matching close query tag (think of bookends) we set off what's in between for special treatment: Forge does what's between the open and close query tags once for each record it finds.

Said another way, by pairing the open query tag with a matching close query tag we create a control structure, and what we put inside this structure gets special treatment.

So what about this template:

```
These MDW employees have blue eyes:<P>
     <mtl mdw.personnel eye=blue>
          <mtl mdw.personnel.name>
     </mtl mdw.personnel>
          <mtl mdw.personnel.department><br>
```

Where did the department info go, and why do we get the
 tag only once?:

```
These MDW employees have blue eyes:<P>
Kevyn
Barbara
David
Chris
Wayne
Brent
  <br>
```

We get the
 tag once because it's outside the control structure created by the open/close pair of query tags. And everything outside a control structure only "happens" once.

So, I can hear you asking yourself, "why didn't the <mtl mdw.personnel.department> tag happen?"

We didn't get data from that tag because data tags only have meaning when they're inside a query structure; it's all about context.

For example, suppose I walked up to you and said "subtract two from that." Without context, without knowing what "that" is, you have no idea what I'm talking about. And it's pretty much the same for Forge. A data tag "out of context" (outside of a pair of query tags,) has no meaning.

That's because a data tag is shorthand for something like "give me what's in the [colname] column on this row." And once you're outside the query tags (before the open tag or after the

close) there's no "this row." Since "this row" is blank, the value in the [colname] column of "this row" is blank, and your data tag brings back a blank value.

So, it doesn't matter in the above example that Forge gets to the department data tag right after it leaves a query structure. There's no "this row" to pull the data from, and Forge doesn't "remember" anything from inside the query tags.

Debug Note:

- If your template is only pulling back one record, but you expected more than one, check to see that [1] you have a close query tag and [2] it matches your open query tag (spelling counts!)

If either of these conditions is not true, more than likely Forge found more than the first record, but doesn't know what to do with the rest.

- If your template is giving you data for some tags, but blanks for others, make sure all of your tags are inside your open/close pair of query tags.

Loop-da-loop

Intro to Loops

We've looked already at MTL's most fundamental control structure: the query/close query pair of tags. What's between these tags gets repeated once for each record that matches the query.

The loop control structure is similar to the query control structure: it repeats what's between the open and close tags. How many times it loops through is specified in the open tag.

Loop looks like this:

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

Given that loop, Forge would return this:

hello
hello
hello

Could you just as easily have typed "hello
" three times? Sure. But that's not the point. There will be times when it'll be easier to use a loop than to type it out yourself. (We'll revisit this after we've talked about variables.)

What number, please?

Numbering the Output

In earlier examples, we used data tags (`<mtl category.table.colname>`) to pull out information about the records Forge found. But loop isn't associated with any tables. There's no real data for Forge to spit out… except which time through the loop we're on. To spit that out we use the index tag:

```
<mtl loop times=3>
     <mtl loop.index> hello<br>
</mtl loop>
```

Would spit out this:

```
1 hello<br>
2 hello<br>
3 hello<br>
```

The index tag is actually a universal MTL tag. You can tack ".index" onto any category.table tag and it will number the records Forge finds for that query.

So to revisit our personnel example:

```
<mtl mdw.personnel eye=blue>
     <mtl mdw.personnel.index> <mtl mdw.personnel.name>
          <mtl mdw.personnel.department><br>
</mtl mdw.personnel>
```

We get:

```
1 Kevyn
  support<br>
2 Barbara
  executive<br>
3 David
  NOC<br>
4 Chris
  engineering<br>
5 Wayne
  engineering<br>
6 Brent
  NOC<br>
```

Does that mean there's a column in the `mdw.personnel` table named index? No.

But if there's no "index" column in the table, where did the data come from? There are some pieces of information we can pull out, some data tags, that don't correspond directly to columns in a table; they're calculations, and index is one of them.

Think of a deck of cards: You shuffle and start dealing. The first card off the top is always number 1 - whether it's the ace of spades or the 7 of diamonds. That's how the index tag works.

Sort of…

Sorting with 'orderby'

So at this point, I'm guessing you're wondering why Kevyn gets to be first; why doesn't Forge give us the names in alphabetical order? The answer is: because we didn't ask it to.

Unless it's told otherwise, Forge will return the records that match a query in an arbitrary order.

To change that, use the "`orderby=colname`" query qualifier:

```
<mtl mdw.personnel eye=blue orderby=name>
     <mtl mdw.personnel.index> <mtl mdw.personnel.name>
          <mtl mdw.personnel.department><br>
</mtl mdw.personnel>
```

Changes our results to:

```
1 Barbara
  executive<br>
2 Brent
  NOC<br>
3 Chris
  engineering<br>
4 David
  NOC<br>
5 Kevyn
  support<br>
6 Wayne
  engineering<br>
```

Can you sub-order? Yes.:

```
<mtl mdw.personnel eye=blue orderby=department,name>
  <mtl mdw.personnel.index> <mtl mdw.personnel.name>
          <mtl mdw.personnel.department><br>
</mtl mdw.personnel>
```

The primary order this time is department, so the folks in engineering come up first, even thought their names aren't first in the alphabet. But within engineering, the names are alphabetical:

```
1 Chris
     engineering<br>
2 Wayne
     engineering<br>
3 Barbara
     executive<br>
4 Brent
```

```
     NOC<br>
5 David
     NOC<br>
6 Kevyn
     support<br>
```

To invert the order, use this format: `orderby=colname:d` (d for **d**escending.):

```
 <mtl mdw.personnel eye=blue orderby=department:d,name>
     <mtl mdw.personnel.index> <mtl mdw.personnel.name>
          <mtl mdw.personnel.department><br>
</mtl mdw.personnel>
```

Output:

```
1 Kevyn
  support<br>
2 Brent
  NOC<br>
3 David
  NOC<br>
4 Barbara
  executive<br>
5 Chris
  engineering<br>
6 Wayne
  engineering<br>
```

Notice that the order got inverted *only* for department. Within departments, names were still in alphabetical order.

- *Generally* you can order a query by any column in the table.
- You can order a query by as many or as few columns as you want.
- You can order each column ascending (the default) or descending (with ":d" or ":D") independently of the others.

A little iffy

Intro to the If Statement

We've looked at two control structures so far: the query and the loop. They're both special instructions to Forge *not* to go from the top of the template straight to the bottom. MTL has one more control structure: the if statement. While the query and loop statements set up a repeat pattern, an if statement is like a fork in the road.

Two roads diverged in a wood and I - I took the one less traveled by…

This should come pretty naturally to most folks: *If* it's a work-night, set the alarm for 6 a.m.; *else* set it for 9 a.m. *If* it's cold outside, wear a coat; *else* don't.

Okay, so how do we say that in MTL?

```
<mtl if {[test]}>
     [response if the test is true]
<mtl else>                        optional
     [alternate response]               optional
</mtl if>
```

The things we can test for:

- `=` equality
- `>` greater than
- `>=` greater than or equals
- `<` less than
- `<=` less than or equals
- `!=` non-equality

There are a few more things we can test for, but we'll cover those later. (In *Fun with strings*.)

So let's put it together:

```
<mtl mdw.personnel eye=blue>
     <mtl if {mdw.personnel.department = "engineering"}>
          <mtl mdw.personnel.name> <mtl mdw.personnel.department> rocks!
     </mtl if>
</mtl mdw.personnel>
```

This is what we get back:

```
Chris engineering rocks!
Wayne engineering rocks!
```

What happens for the records where department doesn't match "engineering"? Nothing. We don't even get the name and department. Maybe we should add an else:

```
<mtl mdw.personnel eye=blue>
     <mtl if {mdw.personnel.department = "engineering"}>
          <mtl mdw.personnel.name><mtl mdw.personnel.department> rocks!
     <mtl else>
          <mtl mdw.personnel.name><mtl mdw.personnel.department>
     </mtl if>
</mtl mdw.personnel>
```

So now we get the name and department for everyone:

```
Kevyn support
Barbara executive
David NOC
Chris engineering rocks!
Wayne engineering rocks!
Brent NOC
```

But… But it seems silly to type in the name and department data tags twice – silly *and* like a lot of extra typing! The fact is, we want the name and department for everyone where `eye=blue`. It's just that we want to do something extra for folks in engineering.

So, we *always* want name and department. In either case (whether mdw.personnel.department = "engineering" or not) we want to print those things out. So rather than adding an else clause to the if statement to handle the non-engineers, it makes sense to just move the stuff we *always* want to somewhere outside the if statement:

```
<mtl mdw.personnel eye=blue>
     <mtl mdw.personnel.name><mtl mdw.personnel.department>
     <mtl if {mdw.personnel.department = "engineering"}>
           rocks!
     </mtl if>
</mtl mdw.personnel>
```

Notice we've eliminated the need for an else clause:

```
Kevyn support
Barbara executive
David NOC
Chris engineering
  rocks!
Wayne engineering
  rocks!
Brent NOC
```

And we've made a better template. Now it's easier to read. That makes it easier to deal with if there are problems (i.e. it makes it easier to debug.) And our new template is also easier to

maintain. For instance, if you wanted to start printing out hair color for all your blue-eyed folks, you'd have two edits to make in the "bad" template :

```
<mtl mdw.personnel eye=blue>
     <mtl if {mdw.personnel.department = "engineering"}>
          <mtl mdw.personnel.name> <mtl mdw.personnel.hair>
          <mtl mdw.personnel.department> rocks!
     <mtl else>
          <mtl mdw.personnel.name> <mtl mdw.personnel.hair>
          <mtl mdw.personnel.department>
     </mtl if>
</mtl mdw.personnel>
```

But with the "good" way, you only have one edit to make:

```
<mtl mdw.personnel eye=blue>
     <mtl mdw.personnel.name> <mtl mdw.personnel.hair>
     <mtl mdw.personnel.department>
     <mtl if {mdw.personnel.department = "engineering"}>
           rocks!
     </mtl if>
</mtl mdw.personnel>
```

Of course, when we switched from the "bad" template to the "good" template, the picky among you will have noticed that the tabs and returns in the output changed from one version to the other. If you're sending your output to a browser though, it will ignore the extra linebreaks and tabs. If you're not; if you're in a situation where those extra spaces matter, perhaps the first version is what you really need.

But in general, it's good practice to put the things you want to *always* happen, no matter what, *outside* your if/else structure.

Let's be logical about this…

Testing with "and" and "or"

So… What if we want to test multiple conditions? We use what the engineers call "logical operators": and & or.

We'll leave out the query qualifier this time so we have more results to work with:

```
<mtl mdw.personnel>
     <mtl mdw.personnel.name>
          <mtl if {mdw.personnel.eye = "blue" and
                        mdw.personnel.hair != "brown"}>
               is fair skinned!
     </mtl if>
</mtl mdw.personnel>
```

Only Barbara (blue eyes, blond hair) and David (blue eyes, red hair) get labeled fair skinned:

```
Kevyn
Jerome
Natasha
Barbara
  is fair skinned!
David
  is fair skinned!
Michael
Felicia
Toby
Chris
Wayne
Nik
Angelia
Connie
Brent
Bill
```

"Or" works like you might expect:

```
<mtl mdw.personnel>
     <mtl mdw.personnel.name>
     <mtl if {mdw.personnel.eye != "brown" or
                    mdw.personnel.hair = "blond"}>
          is fair skinned!
     </mtl if>
</mtl mdw.personnel>
```

Notice, more people are labeled fair skinned this time because the "or" test is true for people whose eyes are not brown and people whose hair is blond. So folks with blue or green or grey eyes get labeled fair skinned as do brown-eyed folk with blond hair:

```
Kevyn
  is fair skinned!
Jerome
Natasha
Barbara
  is fair skinned!
David
  is fair skinned!
Michael
Felecia
  is fair skinned!
Toby
  is fair skinned!
Chris
  is fair skinned!
Wayne
  is fair skinned!
Nik
Angelia
  is fair skinned!
Connie
Brent
  is fair skinned!
Bill
  is fair skinned!
```

Multiple logical operators

Testing with multiple "and" and "or" operators

If you're only testing two conditions (A *and* B … or … A *or* B) you're good to go.

If you're testing multiple conditions with the same logical operator (A *and* B *and* C … or … A *or* B *or* C) you're similarly good to go.

But if you're mixing logical operators (A *and* B *or* C) you'll need to use parentheses to make sure Forge comes up with the results you expect it to:

i.e. if you give Forge this

```
A and B or C
```

It could mean

```
(A and B) or C
```

or:

```
A and (B or C)
```

If A is false and B and C are true, then the first interpretation [(A and B) or C] would give you true while the second [A and (B or C)] would give you false.

And it's hard to predict what Forge will think it means and which conditions it'll evaluate to true:

If Forge thinks 'A and B or C' means:	It'll do your 'if' if these things are true:
A and (B or C)	A & C
A and (B or C)	A & B
(A and B) or C	A & B
(A and B) or C	C

Since you can't predict what Forge will think your expression means, you must use parentheses or suffer the consequences: hours of debugging a template that doesn't work because Forge thinks 'A and B or C' means '(A and B) or C' but you thought it meant 'A and (B or C)' or vice versa.

So to head off any confusion, use parentheses:

```
<mtl mdw.personnel>
     <mtl mdw.personnel.name>
     <mtl if {mdw.personnel.hair = "brown" and (mdw.personnel.eye =
"blue" or mdw.personnel.eye green")}>
               light-eyed brunette!
```

```
    </mtl if>
</mtl mdw.personnel>
```

Labels folks with brown hair and blue eyes and folks with brown hair and green eyes as light-eyed brunettes:

```
Kevyn
  light-eyed brunette!
Jerome
Natasha
Barbara
David
Michael
Felicia
  light-eyed brunette!
Toby
Chris
  light-eyed brunette!
Wayne
  light-eyed brunette!
Nik
Angelia
Connie
Brent
  light-eyed brunette!
Bill
```

Variables

So far we've worked mainly with data stored in tables in the database (remember, index is a calculation.)

It is possible, however, to store data temporarily in what's called a "variable." It's called a variable because what's inside it is "likely to change or vary; subject to variation; changeable."*

A variable is a temporary container for data (think of scratch paper.) Putting a value in a variable writes it on your scratch paper, setting that value aside for later use. With scratch paper, you get back the information you saved by picking up the scratch paper and reading it. It works the same way with variables: to save a value, you write, or store it in your variable. To get it back out, you pick up your variable and read the contents.

Here's the syntax to store something in, or assign a value to, a variable:

```
<mtl $name="G.Ann">             variable assignment
```

To get that value back, or reference it, you use just the variable name:

```
<mtl $name>                           variable reference
```

And Forge responds with:

```
G.Ann
```

You can change the value in a variable by just re-assigning it:

```
<mtl $name="G.Ann">             variable assignment
<mtl $name><br>                       variable reference
<mtl $name="Wayne">             variable re-assignment
<mtl $name><br>                       variable reference
```

And Forge responds with:

```
G.Ann<br>
Wayne<br>
```

You can re-assign the value of a variable or read it out as many or as few times as you like:

```
<mtl $name="G.Ann">             variable assignment
<mtl $name="Wayne">             variable assignment
<mtl $name><br>                       variable reference
<mtl $name><br>                       variable reference
```

Notice, we stored two different values in $name before we ever referenced it:

```
Wayne<br>
Wayne<br>
```

What happened to the "G.Ann" value we stored in `$name` originally? It got tossed out when we assigned "Wayne" to `$name`.

So let's combine variables with query and data tags:

```
<mtl mdw.personnel>
     <mtl mdw.personnel.name>
     <mtl if {mdw.personnel.eye != "brown"}>
          <mtl $lightEyed = mdw.personnel.name>
     </mtl if>
</mtl mdw.personnel>
lightEyed: <mtl $lightEyed>
```

In this example `$lightEyed` gets assigned 9 times.:

```
Kevyn                    $lightEyed is Kevyn
Jerome                   $lightEyed is Kevyn
Natasha                  $lightEyed is Kevyn
Barbara                  $lightEyed is Barbara
David                    $lightEyed is David
Michael                  $lightEyed is David
Felicia                  $lightEyed is Felicia
Toby                     $lightEyed is Toby
Chris                    $lightEyed is Chris
Wayne                    $lightEyed is Wayne
Nik                      $lightEyed is Wayne
Angelia                  $lightEyed is Wayne
Connie                   $lightEyed is Wayne
Brent                    $lightEyed is Brent
Bill                     $lightEyed is Bill
LightEyed: Bill
```

But only the last value it got before being referenced shows up here.

All variable names start with a dollar sign. Letters, numbers and underscores are "legal" characters in a variable name, but the first character after the dollar sign must be a letter.

"Legal" (i.e. okay) variable names:

```
$var
$var1
$different_var
```

Illegal variable names:

```
$1var
$different-var
$var.name
```

* From Dictionary.com.

Syntax side-trip

Spaces, Literal Text and Variables

So you've seen how to assign literal values to a variable:

```
<mtl $name = "Wayne">
```

The quotes are actually optional in this case, but would be required if the value we were trying to put in the variable had spaces or angle brackets ("<" or ">") or a few other special characters:

```
<mtl $name="G. Ann Campbell">
```

versus

```
<mtl $name=G. Ann Campbell>
```

Without the quotes, Forge stops at the first space (`$name` gets "G."), then doesn't know what to do with the rest & throws out an error message.

You've seen how to store the value of a data tag to a variable:

```
<mtl $name = mdw.personnel.name>
```

How would you assign literal text and the value of a data tag to a variable? Like so:

```
<mtl $name = "{mdw.personnel.name} rocks!">
```

In this case the quotes are required because we're mixing types of data (we've got literal text and a data tag) in our assignment.

Additionally, curly braces are required around the data tag. You've seen curly braces before in if statements; they tell Forge to evaluate what's inside and use the result.

You'll also need curly braces if you're assigning multiple data tags to a variable:

```
<mtl mdw.personnel>
     <mtl $name = {mdw.personnel.name}{mdw.personnel.department}>
</mtl mdw.personnel>
<mtl $name>
```

Notice, that name and department are slammed together in the result:

```
Billsales
```

Put a space between the data tags:

```
<mtl mdw.personnel>
     <mtl $name = {mdw.personnel.name} {mdw.personnel.department}>
</mtl mdw.personnel>
<mtl $name>
```

And Forge gets "confused" and gives you an error. So, how to get a space between "Bill" and "sales"? You're combining literal text (the space) with symbolic values (variables and/or data tags), so you'll need to surround the whole thing with quotes:

```
<mtl mdw.personnel>
     <mtl $name = "{mdw.personnel.name} {mdw.personnel.department}">
</mtl mdw.personnel>
<mtl $name>
```

And you get back:

Bill sales

That means this code:

```
<mtl $name = "mdw.personnel.name rocks!">
<mtl $name>
```

Would return this:

mdw.personnel.name rocks!

Putting it together

If Statement Imbedded in a Query

Let's look at more complicated examples. So far we've only looked at the data in four columns of the `mdw.personnel` table, but there are more. We'll add the years column to give us some numbers to work with:

name	eye	hair	department	years
Kevyn	blue	brown	support	4
Jerome	brown	salt & pepper	support	3
Natasha	brown	black	support	2
Barbara	blue	blond	executive	7
David	blue	red	NOC	2
Michael	brown	salt & pepper	executive	7
Felecia	green	brown	content	1
Toby	green	blue	content	3
Chris	blue	brown	engineering	7
Wayne	blue	brown	engineering	6
Nik	brown	black	content	2
Angelia	brown	blond	content	1
Connie	brown	black	sales	0.5
Brent	blue	brown	NOC	2
Bill	grey	none	sales	5

Let's run through every record in the table:

```
<mtl mdw.personnel>
     <mtl if {mdw.personnel.years > $longest}>
          <mtl $name=mdw.personnel.name>
          <mtl $longest=mdw.personnel.years>
     </mtl if>
</mtl mdw.personnel>
<mtl $name> has been with MDW for <mtl $longest> years.
```

For each record, we compare the years to the value of `$longest`. But what happens the first time through the loop, before `$longest` has been set to anything?

All variables have blank values until you give them a value. So the first time you test `$longest`, `$longest` is an empty string. And the numeric value of an empty string is 0.

Here's what happens, row by row:

```
years=4. 4>0. $longest=4. $name=Kevyn.
years=3. 3<4. $longest=4. $name=Kevyn.
years=2. 2<4. $longest=4. $name=Kevyn.
years=7. 7>4. $longest=7. $name=Barbara.
```

```
years=2. 2<7. $longest=7. $name=Barbara.
years=7. 7=7. $longest=7. $name=Barbara.
years=1. 1<7. $longest=7. $name=Barbara.
years=3. 1<7. $longest=7. $name=Barbara.
years=7. 7=7. $longest=7. $name=Barbara.
years=6. 6<7. $longest=7. $name=Barbara.
years=2. 2<7. $longest=7. $name=Barbara.
years=1. 1<7. $longest=7. $name=Barbara.
years=.5. .5<7. $longest=7. $name=Barbara.
years=2. 2<7. $longest=7. $name=Barbara.
years=5. 5<7. $longest=7. $name=Barbara.
Barbara has been with MDW for 7 years.
```

Remember, `$longest` starts out essentially at 0, so the first person we look at, Kevyn, automatically has the longest time in service so far. So `$longest` gets Kevyn's years, 4, and `$name` gets "Kevyn."

The value in the years column in Jerome's row is 3, which is smaller than 4, so we do nothing.

The same for Natasha.

Then we get to Barbara. Her years value, 7, is greater than `$longest`, so we reassign `$longest` to Barbara's years, 7, and reassign `$name` to "Barbara."

No other row has a years value greater than 7, so `$name` stays "Barbara."

But we can look at the table and see that Barbara is not the only one with 7 years' service; she just happens to be the first one.

So let's try it again. This time we'll keep a list of folks with a long time in service:

```
<mtl mdw.personnel>
     <mtl if {mdw.personnel.years > $longest}>
          <mtl $name=mdw.personnel.name>
          <mtl $longest=mdw.personnel.years>
     <mtl else>
          <mtl if {mdw.personnel.years = $longest}>
               <mtl $namelist="{$namelist},
                     {mdw.personnel.name}">
          </mtl if>
     </mtl if>
</mtl mdw.personnel>
<mtl $name><mtl $namelist> have been with MDW for
     <mtl $longest> years.
```

And we get:

```
Barbara, Michael, Chris have been with MDW for
  7 years.
```

Here's what happens, row by row. This time we'll only show variables when they change:

```
years=4. 4>0. $longest=4. $name=Kevyn.
years=3. 3<4.
```

```
years=2. 2<4.
years=7. 7>4. $longest=7. $name=Barbara.
years=2. 2<7.
years=7. 7=7. $longest=7. $name=Barbara.
  $namelist=", Michael".
years=1. 1<7.
years=3. 1<7.
years=7. 7=7. $longest=7. $name=Barbara.
  $namelist=", Michael, Chris".
years=6. 6<7.
years=2. 2<7.
years=1. 1<7.
years=0.5. 0.5<7.
years=2. 2<7.
years=5. 5<7.
```

This time when we come to Michael, we see that his years value is 7, the same as $longest. The first if test is false: 7 is not larger than 7, so we go to the else clause. In the else clause we see that the years value is the same as $longest, so we set `$namelist` to: itself plus a comma, plus "Michael." That puts an extra comma at the beginning of `$namelist` because `$namelist` starts out blank.

When we get to the 3rd person with 7 years' service, Chris, we set `$namelist` to itself, plus a comma, plus "Chris" and end up with ", Michael, Chris".

A little housekeeping

Learning to Make Better Queries

What happens if we order the query from the last example by years?

```
<mtl mdw.personnel orderby=years>
     <mtl if {mdw.personnel.years > $longest}>
          <mtl $name=mdw.personnel.name>
          <mtl $longest=mdw.personnel.years>
     <mtl else>
          <mtl if {mdw.personnel.years = $longest}>
               <mtl $namelist="{$namelist},
                     {mdw.personnel.name}">
          </mtl if>
     </mtl if>
</mtl mdw.personnel>
<mtl $name><mtl $namelist> have been with MDW for
<mtl $longest> years.
```

We get:

```
years=.5 .5>0 $longest=.5 $name=Connie
years=1 1>.5 $longest=1 $name=Felicia
years=1 1=1 $longest=1 $name=Felicia
  $namelist=", Angelia"
years=2 2>1 $longest=2 $name=Natasha
  $namelist=", Angelia"
years=2 2=2 $longest=2 $name=Natasha
  $namelist=", Angelia, David"
years=2 2=2 $longest=2 $name=Natasha
  $namelist=", Angelia, David, Nik"
years=2 2=2 $longest=2 $name=Brent
  $namelist=", Angelia, David, Nik, Brent"
years=3 3>2 $longest=3 $name=Jerome
  $namelist=", Angelia, David, Nik, Brent"
years=3 3=3 $longest=3 $name=Jerome
  $namelist=", Angelia, David, Nik, Brent, Toby"
years=4 4>3 $longest=3 $name=Kevyn
  $namelist=", Angelia, David, Nik, Brent, Toby"
years=5 5>4 $longest=5 $name=Bill
  $namelist=", Angelia, David, Nik, Brent, Toby"
years=6 6>5 $longest=6 $name=Wayne
  $namelist=", Angelia, David, Nik, Brent, Toby"
years=7 7>6 $longest=7 $name=Barbara
  $namelist=", Angelia, David, Nik, Brent, Toby"
years=7 7=7 $longest=7 $name=Barbara
  $namelist=", Angelia, David, Nik, Brent, Toby, Michael"
years=7 7=7 $longest=7 $name=Barbara
  $namelist=", Angelia, David, Nik, Brent, Toby, Michael, Chris"
Barbara, Angelia, David, Nik, Brent, Toby, Michael, Chris have been with MDW for 7 years.
```

Obviously, we have a problem; Angelia has not yet been with MDW for 7 years. Nor have David, Nik, Brent and Toby. Why did this show up when we ordered the query but not when we didn't? We got lucky the first time.

But the problem's not hard to solve: reset `$namelist` to blank each time `$name` and `$longest` are reassigned:

```
<mtl mdw.personnel orderby=years>
     <mtl if {mdw.personnel.years > $longest}>
          <mtl $name=mdw.personnel.name>
          <mtl $longest=mdw.personnel.years>
          <mtl $namelist="">
     <mtl else>
          <mtl if {mdw.personnel.years = $longest}>
               <mtl $namelist="{$namelist},
                     {mdw.personnel.name}">
          </mtl if>
     </mtl if>
</mtl mdw.personnel>
<mtl $name><mtl $namelist> have been with MDW for
<mtl $longest> years.
```

This time we get:

```
years=.5 .5>0 $longest=.5 $name=Connie
years=1 1>.5 $longest=1 $name=Felicia
years=1 1=1 $longest=1 $name=Felicia $namelist=", Angelia"
years=2 2>1 $longest=2 $name=Natasha $namelist=""
years=2 2=2 $longest=2 $name=Natasha $namelist=", David"
years=2 2=2 $longest=2 $name=Natasha $namelist=", David, Nik"
years=2 2=2 $longest=2 $name=Natasha $namelist=", David, Nik, Brent"
years=3 3>2 $longest=3 $name=Jerome $namelist=""
years=3 3=3 $longest=3 $name=Jerome $namelist=", Toby"
years=4 4>3 $longest=3 $name=Kevyn $namelist=""
years=5 5>4 $longest=5 $name=Bill $namelist=""
years=6 6>5 $longest=6 $name=Wayne $namelist=""
years=7 7>6 $longest=7 $name=Barbara $namelist=""
years=7 7=7 $longest=7 $name=Barbara $namelist=",       Michael"
years=7 7=7 $longest=7 $name=Barbara $namelist=", Michael, Chris"
Barbara, Michael, Chris have been with MDW for 7 years.
```

Loopy

More about Loops

Now that we've covered variables, let's revisit the loop control structure. The loop example we used:

```
<mtl loop times=3>
     hello
</mtl loop>
```

Begs the question: wouldn't it be simpler to type it out yourself:

hello
hello
hello

So let's look at an example where it isn't easier to type it out yourself. This time, we're going to use the value of `$longest` to specify the number of loop iterations :

```
<mtl mdw.personnel orderby=years>
     <mtl if {mdw.personnel.years > $longest}>
          <mtl $name><mtl $namelist> have been with MDW for
          <mtl loop times=$longest>
               years
          </mtl loop>
          <mtl $name=mdw.personnel.name>
          <mtl $longest=mdw.personnel.years>
          <mtl $namelist="">
     <mtl else>
          <mtl if {mdw.personnel.years = $longest}>
               <mtl $namelist="{$namelist},
                     {mdw.personnel.name}">
          </mtl if>
     </mtl if>
</mtl mdw.personnel>
```

Note, that the loop doesn't kick in for Connie; $longest is .5 for her, and numbers to the right of the decimal get ignored in this context (0.5 becomes 0), so the loop runs 0 times for her:

Connie have been with MDW for
Felicia, Angelia have been with MDW for years
Natasha, David, Nik, Brent have been with MDW for years years
Jerome, Toby have been with MDW for years years years
Kevyn have been with MDW for years years years years
Bill have been with MDW for years years years years years
Wayne have been with MDW for years years years years years years
Barbara, Michael, Chris have been with MDW for years years years years years years years

We've ordered the query results by years this time, so each time `mdw.personnel.years` is larger than `$longest`:

```
<mtl if {mdw.personnel.years > $longest}>
```

we know we've come to the end of the folks that have the current value of $longest in the years column. I.e. the list of people with that many years is complete. So we output the list

```
<mtl $name><mtl $namelist> have been with MDW for
<mtl loop times=$longest>
     years
</mtl loop>
```

before we reassign the variables.

```
<mtl $name=mdw.personnel.name>
<mtl $longest=mdw.personnel.years>
<mtl $namelist="">
```

Fun with strings

String Functions in MTL

Whether you realized it or not, everything we've dealt with in MTL has been a string.

No, it hasn't come off a ball of twine, it's been "a set of consecutive characters."*

Like beads on an actual string, every piece of data MTL has given us has been a series of characters: letters, numbers, spaces and punctuation. (Each key on your keyboard produces a character. Yes, space, return and tab count as characters just like the letters and numbers.)

And just like beads on a thread, a string of characters has a beginning and an end. And like those beads, we can count and number the characters in a string.

We'll start with a variable:

```
<mtl $testVar="This is a sentence.">
```

Here's how Forge sees it:

```
var name       1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
$testVar = "  T  h  i  s     i  s     a     s  e  n  t  e  n  c  e  .  "
```

Notice that the quote marks didn't get numbered; they're not part of the string, they just mark the beginning and end. Notice also that the spaces and period did get numbered.

So, now you're wondering what good this is, right?

Now that we know about the underlying structure of strings, we'll be able to use MTL's string functions. Note that we'll mainly be using variables in the examples below, but you can also use data tags or plain old text (if you put it between quotes.)

String functions that report:

First, let's talk about the string functions that tell you about your string:

- `STRING.LENGTH(text)` - Returns the number of characters in "text".

  So:

  ```
  <mtl string.length($testVar)>
  ```

  Returns:

19

- STRING.CONTAINS(text1,text2) - Returns the position of the character in text1 where text2 starts. 0 means "not found," text2 is not in text1.

  So:

  ```
  <mtl string.contains($testVar,"sentence")>
  ```

  Returns:

  11

  But because "blue" is not in the string "This is a sentence.", this:

  ```
  <mtl string.contains($testVar,"blue")>
  ```

  gives you:

  0

- `STRING.INLIST(list,item)` - Returns 1 (true) if "item" is a member of the comma-separated "list," otherwise it returns 0 (false).

  You'll probably use this function mainly in if statements. Without `string.inlist`:

  ```
  <mtl mdw.personnel>
       <mtl mdw.personnel.name>
       <mtl if {mdw.personnel.hair = "brown" or
            mdw.personnel. hair = "salt & pepper" or
            mdw.personnel.hair black")}>
            brunette!
       </mtl if>
       <br>
  </mtl mdw.personnel>
  ```

  With string.inlist:

  ```
  <mtl $hair="brown,black,salt & pepper">
  <mtl mdw.personnel>
       <mtl mdw.personnel.name>
       <mtl if {string.inlist($hair,mdw.personnel.hair)>0}>
            brunette!
       </mtl if>
       <br>
  </mtl mdw.personnel>
  ```

  String.inlist is preferred over string.contains when [1)] you're actually dealing with a comma-separated list, and [2)] string.contains might give you false positives:

  ```
  <mtl $id_numbers="986,423,1325,4">
  <mtl if {string.contains($id_numbers,1)>0}>
  ```

```
 "<mtl $id_numbers>" contains 1.<br>
<mtl else>
 "<mtl $id_numbers>" does NOT contain 1.<br>
</mtl if>

<mtl if {string.inlist($id_numbers,1)>0}>
 1 is an item in this list: <mtl $id_numbers><br>
<mtl else>
 1 is NOT an item in this list: <mtl $id_numbers><br>
</mtl if>
```

Gives us:

```
"986,423,1325,4" contains 1.<br>  false positive?
1 is NOT an item in this list: 986,423,1325,4<br>
```

Notice that to use the string functions in if statements, we tested their results against zero. A zero result means "false;" one string is not in the other. Some number greater than zero means "true."

String functions that manipulate:

Now that we know all about our string, let's do something with them:

- `STRING.SUBSTR(text, start_pos)` - Returns "text" starting at "start_pos".
- `STRING.SUBSTR(text, start_pos, end_pos)` - Returns "text" starting at "start_pos" and ending at "end_pos".

The substring function lets you snip out pieces of text. You must give it a string to snip from and the position it should start snipping at. You may optionally give it the position where it should end the snippet:

```
<mtl string.substr($testVar,11)>
```

Gives you everything from position 11 to the end of the string:

sentence.

Also:

```
<mtl string.substr($testVar,11,14)>
```

Gives you everything from position 11 to position 14, inclusive:

sent

You could also use a variable or a calculation to specify the starting position:

```
<mtl $pos=string.contains($testVar,"a")>
<mtl string.substr($testVar,$pos)>
```

Gets you:

a sentence.

Variables and/or calculations can also be used for the end position. Note that calculations must be enclosed in curly braces to force Forge to interpret what's inside and use the result:

```
<mtl $pos=string.contains($testVar,"a")>
<mtl string.substr($testVar,$pos,{$pos+5})>
```

Returns:

a sent

- `STRING.URL_ENCODE(text)` - Returns "text" with spaces, dashes &etc. URL-encoded.

  For instance, this:

  ```
  <mtl $urlbit="augusta ga">
  <mtl string.url_encode($urlbit)>
  ```

  Turns into this:

  augusta%20ga

  But be aware that this:

  ```
  <mtl $urlbit="property=augusta ga">
  <mtl string.url_encode($urlbit)>
  ```

  Turns into this:

  property%3Daugusta%20ga

Examples:

Now that we have string functions in the tool box, we can "tighten up" the years-of-service example we did earlier:

```
<mtl mdw.personnel orderby=years>
     <mtl if {mdw.personnel.years > $longest}>
          <mtl $name=mdw.personnel.name>
          <mtl $longest=mdw.personnel.years>
     <mtl else>
```

```
          <mtl if {mdw.personnel.years = $longest}>
               <mtl $name="{$name}, {mdw.personnel.name}">
          </mtl if>
     </mtl if>
</mtl mdw.personnel>
<mtl $name>
<mtl if {string.contains($name,",")>0}>
     have
<mtl else>
     has
</mtl if>
been with MDW for <mtl $longest> years.
```

This time we get:

```
Barbara, Michael, Chris
  have
been with MDW for 7 years.
```

But if there were only one name in $name, we'd get:

```
Barbara
  has
been with MDW for 7 years.
```

That example only uses the "reporting" string functions. Let's give the "manipulation" string functions a workout. We'll try modifying a string by bolding its first three words.

It's easy enough to start the bolding; you just output `<b>` tag before you output the string. But were to end the bolding? And how do we insert the `</b>` tag into our string? Those are the challenges of this exercise.

We'll meet those challenges by chopping our target string into two substrings: the substring to be bolded and the part we don't want bold. We'll call those two variables, for hopefully obvious reasons, `$boldstr` (what gets bolded) and `$therest` (what doesn't).

In this example we'll just type in our string so we have something to work with, but we could just as easily put the value of a data tag in `$therest` instead. In fact, that's probably where your string will come from nine times out of ten.

We start by putting the entire string in `$therest` - remember, it holds the non-bold part of our output, and when we start, none of the string is bold yet. And since nothing's bold yet, our second line will (re)set `$boldstr` to null (`""`).

```
<mtl $therest = "This is a test of substringing and indexing.">
<mtl $boldstr="">
```

Why do we bother setting `$boldstr=""`? Don't variables start out empty? They absolutely do – the first time you use them. But remember I said you'll most often use the example we're building here (or something like it) with a data tag - and of course, data tags only have value inside queries. And that implies that this routine will run multiple times in a row.

So imagine what would happen if we didn't reset $boldstr each time. On the first record, we'd end up with $boldstr having the first three words of the data tag. On the second record, $boldstr would have the first three words from record one and the first the words from record two, and so on…

We're looking for the first three word, and words are separated by spaces, so we start by finding the position of the first space:

```
<mtl $spacePos={string.contains($therest," ")}>
```

Now we know where the first word ends. Let's grab it and store it in `$boldstr`:

```
<mtl $boldstr = string.substr($therest,1,$spacePos)>
```

At this point, the first word shows up in both `$boldstr` and `$therest`. We'll take care of that with another substring operation, but first, let's adjust where we're substringing from:

```
<mtl $spacePos={$spacePos+1}>
```

We add one to `$spacePos` because our first substring operation grabbed from position 1 to `$spacePos` *inclusive* – so we got the first word and the space after it in `$boldstr`. We don't want to repeat that space, so we increment (add one to) `$spacePos` so our next substring operation will start *right after* the space:

```
<mtl $therest={string.substr($therest,$spacePos)}>
```

So now we've "moved" the first word from $therest into $boldstr. We're ready to output, right?:

```
<b><mtl $boldstr></b><mtl $therest>
```

And that returns:

This is a test of substringing and indexing.

Oh yeah… we wanted to do the first *three* words. Hmmm… Maybe if we added a loop:

```
<mtl $therest = "This is a test of substringing and indexing.">
<mtl $boldstr="">
<mtl loop times=3>
     <mtl $spacePos={string.contains($therest," ")}>
     <mtl $boldstr = string.substr($therest,1,$spacePos)>
     <mtl $spacePos={$spacePos+1}>
     <mtl $therest={string.substr($therest,$spacePos)}>
</mtl loop>
<b><mtl $boldstr></b><mtl $therest>
```

Crud. That gives us this:

a test of substringing and indexing.

Probably because `<mtl $boldstr>` got reset each time through the loop to be just the new first word of `<mtl $therest>`. So if we change this:

```
<mtl $boldstr = string.substr($therest,1,$spacePos)>
```

To this:

```
<mtl $boldstr = "{$boldstr}{string.substr($therest,1,$spacePos)}">
```

Then the new template:

```
<mtl $therest = "This is a test of substringing and indexing.">
<mtl $boldstr="">
<mtl loop times=3>
     <mtl $spacePos={string.contains($therest," ")}>
     <mtl $boldstr = "{$boldstr}{string.substr($therest,1,$spacePos)}">
     <mtl $spacePos={$spacePos+1}>
     <mtl $therest={string.substr($therest,$spacePos)}>
</mtl loop>
<b><mtl $boldstr></b><mtl $therest>
```

Outputs this:

This is a test of substringing and indexing.

Yay!

* From Dictionary.com

Advanced sorting

Explicit sorting

So far you can have Forge spit out the results of your query in [1)] an arbitrary order [2)] ascending order or [3)] descending order.

But there will be times you want more complete control.

Let's say we want to sort by eye color, but have the green-eyed folks come out first, instead of in the middle of the alphabet. We could do that with a complicated set of if statements and variables.

Or we could use do some special sorting:

```
<mtl mdw.personnel orderby=eye(green)>
     <mtl mdw.personnel.name><mtl mdw.personnel.eye><br>
</mtl mdw.personnel>
```

That brings the green-eyed folk to the top, but leaves the rest in arbitrary order:

```
Felicia green<br>
Toby green<br>
Kevyn blue<br>
Jerome brown<br>
Natasha brown<br>
Barbara blue<br>
David blue<br>
Michael brown<br>
Chris blue<br>
Wayne blue<br>
Nik brown<br>
Angelia brown<br>
Connie brown<br>
Brent blue<br>
Bill grey<br>
```

Essentially, `orderby=eye(green)` tells Forge to sort the query results by green and non-green, and spit out first the records where `eye=green`.

Conversely, this:

```
<mtl mdw.personnel orderby=eye(green):d>
     <mtl mdw.personnel.name><mtl mdw.personnel.eye><br>
</mtl mdw.personnel>
```

Tells Forge to sort the records by green and non-green and spit out *last* the records where `eye=green`:

```
Kevyn blue<br>
```

```
Jerome brown<br>
Natasha brown<br>
Barbara blue<br>
David blue<br>
Michael brown<br>
Chris blue<br>
Wayne blue<br>
Nik brown<br>
Angelia brown<br>
Connie brown<br>
Brent blue<br>
Bill grey<br>
Felicia green<br>
Toby green<br>
```

Either way, to get the non-green-eyed folks sorted by alphabetical eye color, we'll have to do a sub-sort:

```
<mtl mdw.personnel orderby=eye(green),eye>
     <mtl mdw.personnel.name><mtl mdw.personnel.eye><br>
</mtl mdw.personnel>
```

Returns:

```
Felicia green<br>
Toby green<br>
Kevyn blue<br>
Barbara blue<br>
David blue<br>
Chris blue<br>
Wayne blue<br>
Brent blue<br>
Jerome brown<br>
Natasha brown<br>
Michael brown<br>
Nik brown<br>
Angelia brown<br>
Connie brown<br>
Bill grey<br>
```

But what if we want green, then blue, grey and brown? We can put as many values as we want in the sort order parentheses:

```
<mtl mdw.personnel orderby=eye(green,blue,grey,brown)>
     <mtl mdw.personnel.name><mtl mdw.personnel.eye><br>
</mtl mdw.personnel>
```

Here's what we get:

```
Felicia green<br>
Toby green<br>
Kevyn blue<br>
Barbara blue<br>
```

David blue
Chris blue
Wayne blue
Brent blue
Bill grey
Jerome brown
Natasha brown
Michael brown
Nik brown
Angelia brown
Connie brown

Random sorting

Random sorting sounds like an oxymoron, doesn't it? There are times when it's very useful, though.

Here's how you use it:

```
<mtl mdw.personnel orderby=random>
     <mtl mdw.personnel.name><mtl mdw.personnel.eye><br>
</mtl mdw.personnel>
```

And the results you get back will be different each time you run the template (so I won't attempt to show them here.)

You can also do this:

```
<mtl mdw.personnel orderby=eye,random>
     <mtl mdw.personnel.name><mtl mdw.personnel.eye><br>
</mtl mdw.personnel>
```

Which would sort folks by eye color, and bring all the blue-eyed folks to the top of the list. But among the blue-eyed folks the order would be random, and would change each time the template was run.

You can also do this:

```
<mtl mdw.personnel orderby=eye(green,blue,grey,brown),random>
     <mtl mdw.personnel.name><mtl mdw.personnel.eye><br>
</mtl mdw.personnel>
```

Which would perform the explicit eye color sort, and then sub-sort by random.

But you *can't* do this:

```
<mtl mdw.personnel orderby=random,eye(green,blue,grey,brown)>
     <mtl mdw.personnel.name><mtl mdw.personnel.eye><br>
</mtl mdw.personnel>
```

Well, you actually *can* do that, but the results would be practically the same as:

```
<mtl mdw.personnel orderby=eye(green,blue,grey,brown)>
     <mtl mdw.personnel.name><mtl mdw.personnel.eye><br>
</mtl mdw.personnel>
```

Why? Think of a deck of cards. If you separate the cards by suit, then put the suits in order, you're doing `orderby=suit,value`. And the result would be 2-A of Spades, followed by 2-A of Hearts &etc.

If you did `orderby=suit,random`, i.e. sort by suit, then shuffle the suits, you'd still get all the Spades first, but in a random order (one that changed each time you did it - because that's how shuffling works.) Similarly, if you did `orderby=value,random,` you'd get all the 2's first, with the suits in random order, followed by the 3's (suits random,) and so on.

But if you shuffle first and then sort them back out by suit (`orderby=random,suit`), you lose the effect of the shuffling.

Iffy, redux

Old If Syntax versus New If Syntax

The if statement syntax we've looked at is the "new" syntax, but if you're working with existing templates you'll likely encounter the "old" syntax. Both sets of syntax are valid. The old syntax does have a little more functionality than the new.

Here's the syntax you've seen already:

```
<mtl if {mdw.personnel.department = "engineering"}>
```

Here it is in the old syntax:

```
<mtl if val=mdw.personnel.department eq="engineering">
```

Note, there are no curly braces in the old syntax. Each value (whether it's coming out of a variable or a data tag or from text) will be on the right side of an equal sign.

To the left of the first equals sign, you'll almost generally have "val" or "value". To the left of the second one will be your test:

Old Syntax	New Syntax
<mc if val=$a eq=$b>	<mcc if {$a=$b}>
<mcc if val=$a ne=$b>	<mcc if {$a!=$b}>
<mcc if val=$a gt=$b>	<mcc if {$a>$b}>
<mcc if val=$a ge=$b>	<mcc if {$a>=$b}>
<mcc if val=$a lt=$b>	<mcc if {$a<$b}>
<mcc if val=$a le=$b>	<mcc if {$a<=$b}>
<mcc if val=$a contans=$b>	<mcc if {string.contains($a,$b)>0}>
<mcc if val=$a not_contains=$b>	<mcc if {string.contains($a,$b)=0}>
<mcc if val=$a in=$b>	<mcc if {string.inlist($a,$b)=1}>
<mcc if val=$a not_in=$b>	<mcc if {string.inlist($a,$b)=0}>

There are no logical operators (and & or) in the old syntax. To test two conditions, nest if statements. Instead of this:

```
<mtl if {mdw.personnel.eye = "blue" and mdw.personnel.hair != "brown"}>
     is fair skinned!
</mtl if>
```

You do this:

```
<mtl if val=mdw.personnel.eye eq=blue>
     <mtl if val=mdw.personnel.hair ne=brown>
          is fair skinned!
     </mtl if>
```

```
</mtl if>
```

The old if syntax does have two large advantages over the new syntax; you can use it to do date tests and to check for file existence. There are no equivalent s in the new syntax for these tests:

```
<mtl if date=today eq=Thursday>
<mtl if date=today ne=Thursday>
<mtl if date=today before=Friday>
<mtl if date=today after=Friday>
<mtl if file_exists="[domain.com:/path/from/box/root]">
<mtl if file_noexists="[domain.com:/path/from/box/root]">
```

Note that `today` and the days of the week have special meaning in the above if statements; the substitution of `date=` for `val=` flags Forge to do some special date calculations. You can also use the `gt`, `ge`, `lt` and `le` operators with date tests:

```
<mtl if date=today ge=20020822>
<mtl if date=friday lt=20020822>
```

The numeric dates in these tests are in a YYYYMMDD format. There will be more on dates, date formats and date tests later. (These things are covered in "MTL Common tags" in *Time, time, time*; *Storing & manipulating dates*; and *Testing time*.)

Using `date=` in an if statement flags Forge to do date calculations, but this if statement:

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

Would compare the word "today" to "Thursday," and since the words are different, come up with "false."

Note, also, that the `file_exists` test and `file_noexists` tests work (currently) only for files on Morris servers. (Why? Because the mechanism in use is not a web request, but a file system operation: basically, Forge looks in the file system of the server in question, and follows the path to see if there's anything there.)

F.A.Q.s for programmers

Q. Does MTL have an "else if" clause?

A. No. You'll have to nest another if statement inside your else clause.

Q. What's the MTL array syntax?

A. There is none.

Q. What's the data type of a variable in MTL?

A. All MTL variables are stored in strings. When you perform math operations on a variable, an attempt to convert it to a number is made. If the attempt is successful, the operation is performed.

Q. Can I do integer and/or float math on MTL variables?

A. Yes. Modulus (%) and floating point division (/) are both available.

Q. Why doesn't MTL have X feature, like a *real* programming language?

A. Because MTL was never designed to be a "real" programming language. It was designed to allow non-programmers to pull data out of a database.

Q. But what if I *need* X feature to do Y task.

A. If you have access to Task Tracker, put in a task, making your case for the feature; detailing what you need to do, how you've tried to do it and what result you got. If you don't have access to Task Tracker, contact your support representative with the same information. Either your feature request will be put on the wish list for MTL or you'll be given a work-around. Possibly both.