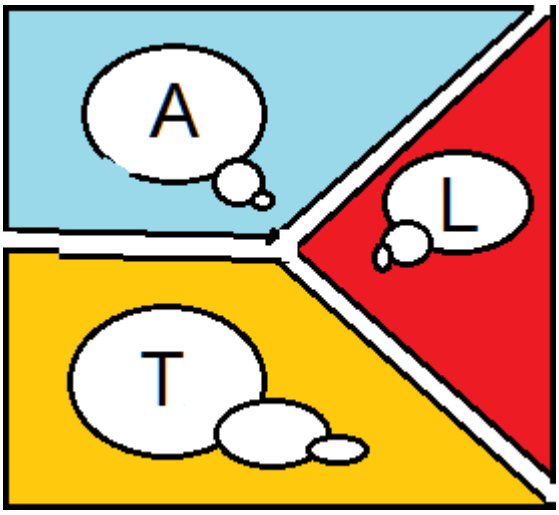
Note: This project has just been started and it merely compiles...

But in case you are interested, here is a brief presentation

j-YATL-on



Yet Another Template Language...

Why YATL?

Most current template languages are procedural, so they look like code, not like template... So they are (at times) hard to understand and need to be debugged, just like code! What about a new *declarative* templating language build from the ground up to look just like a *real* template? Doing so would provide the following benefits:

- **Clarity:** No more `${#if#elseif#endif}` gibberish.
- **Java 8 compatibility:** To accomodate existing projects.
- **Light & simple:** Easy to understand, easy to adapt!
- **Performance**... Whenever possible!
- **No dependency**: Small footprint.

Running the program

- Create a template file containing the following text: `Hello {{$}}!`
- Execute the following *java* code to create a context and invoke the engine:

```
YATL yatl = YATL.getTemplate("filename");
Object root = "World"; // This is the root context
```

```
Writer writer = new StringWriter();
yatl.merge(root, writer); // The same as Velocity!
```

Tutorial

- YATL is intended to be the simplest templating language possible, period.
- As a general rule, values, commands and comments begin and end on the same line.
- Also, be aware that spaces are not allowed inside value expressions, commands and paths.
- YATL is particularly well suited for working with tree data structures and to generate code.

Root context

- The root context aka `$` refers to the object that is provided when launching the template engine. (See running the program above)
- It is recommended that you have full control over the root object so you can implement any special formating services that you may need. YATL will not implement complex computations...

Value

- A value is an expression enclosed in **double braces** `{{ ... }}`. The so called mustache!
- It can be a constant: `{{'any text'}}` or `{{"I'm a text!"}}`
- The result of a value expression is always inserted where it is declared.
- A value expression always starts with any of: the root context, a path or an alias. From there, it is possible to apply any **public** methods that is valid for this object and so on. For exemple you can do `{{$.toString}}` or `{{$.toString.substring(1,4)}}`.
- You can invoke public accessors with `{{$.getSize()}}` or `{{$.getSize}}` or even `{{$.size}}` as you prefer.
- You can use a Map key to access one of its value `{{Map.key}}`.
- If you want to insert any text depending on a value, for example when it is empty or when it is a collection, then assign an alias to this value and enclose it in a block.
- A value expression **MUST NOT** contain spaces. Be sure you remember cause we will not tell you again!

```
{{ $:Root.toString:Alias }} // This is a value
$:Root.toString:Alias       // This is the value expression (no spaces here!)
Root, Alias                 // Well... those are the aliases!
```

Conditional output

- A value can be output conditionnaly by including a test `{{if $.test 'This is true!'}}`.
- To inverse the result of the test, use `!` as in `{{if !$.test 'Oops! Sorry...'}}`.
- You can check if 2 values are equal `{{if $.v1 == $v2 $.v1:ALIAS}}` or not equal `{{if $.v1 != $v2 $.v2:ALIAS}}`.
- In any case, equality is always performed by using the *Java* `==` operator.
- Any `null` object or empty `String` or `Collection` are considered *false* `{{if $ 'Root is not null!'}}`.

Alias

- You may assign an alias (an alternate name) to any part of a value expression. For exemple `{{$:ALIAS_1.toString:ALIAS_2.substring(1,4):ALIAS_3}}`.
- Aliases **MUST** start with a letter or an underscore but they can also contain numbers: `[_A-Za-z][_A-Za-z0-9]*`.
- If any part of a value expression is `null` or empty `""` then no text will be inserted. If you intend to provide a default text in those cases, then you **MUST** provide the value with an alias and put it into a block.
- Any alias outside of a block has no effect.
- Alias can be initialized inside a block `{init $.valueExp:ALIAS}`. An already defined alias is trivial to reuse: `{{ALIAS}}`.

Collection (not yet implemented)

- In case the value is a `Collection`, it can be usefull to know about the *index of* one element or the *size of* the `Collection` itself.
- The `indexOf` function returns the index of the current alias value starting by `1`: `{{indexOf($.val:ALIAS)}}`.
- The `sizeOf` function returns the number of visible alias values: `{{sizeOf($.val:ALIAS)}}`.
- `Collection` functions may be used inside a condition: `{{if indexOf(ALIAS) == 1 'First:'}}`.

Block

- A block always begins with `{begin ALIAS}` and always ends with `{end ALIAS}`.
- A block is always associated to a single alias. Its role is to control the visibilty of the text **surrounding** the values containing this alias.

```
{begin ALIAS} // The most simple block!
    {{$.val:ALIAS}}
{end ALIAS}
```

Empty value

- You may include a `{empty ALIAS}` control followed by the text you want to appear in case the value is empty.

```
{begin ALIAS}
    x = {{$.val:ALIAS}}; // Value Expression: always under {begin}
{empty ALIAS}
    x = null; // Text to appear if the Value is empty
{end ALIAS}
```

Collection value

- When the associated value is a `Collection`, you can optionnaly use any of the following controls **always in this order**: `{before ALIAS}`, `{between ALIAS}`, `{after ALIAS}`, `{empty ALIAS}`.
- All associated alias expressions **MUST** be declared **directly** under the `{begin ALIAS}` control.

```
{begin ALIAS}
{{ALIAS}} // Value Expression: always directly under {begin}
{before ALIAS}
x = new List(" // Text to appear once, before the first value.
{between ALIAS}
"," // Text to appear between each values.
{after ALIAS}
"); // Text to appear once, after the whole Collection.
{empty ALIAS}
x = null; // Text to appear if the Collection is empty.
{end ALIAS}
```

- The previous example (if you remove the java comments) would return `x = new List("a","b");` or `x = null;` in case there is nothing to display.

Short notation (not yet implemented)

- When the `before, between, after, empty` sequences are short, you may use an alternative notation such as `{begin ALIAS |("|","|")|()}` for clarity.
- The first character found after the alias **MUST** be repeated exactly **4 times** in order to be correctly interpreted as the separator. You can use any **single** character that you want for that matter.
- Remember that all control blocks are single lined.

Block imbrication

- Blocks can imbricated but not interlocked.

```
{begin 1}
    {begin 2}{end 2} // This IS valid!
    {begin 3}
{empty 1}
    {end 3} // This is NOT valid!
{end 1}
```

Alias matching

- The same alias may be repeated among many values and paths so they can be matched together. In this case, all of these aliases **MUST** exist and be equal for the corresponding value to be visible.
- It is possible to have optional aliases `{{$.value:?OPTIONAL_ALIAS:ALTERNATE_ALIAS}}` that can be `null` when the others are not. Optional aliases **MUST** have an alternate alias since they can be empty.

```
{begin ALIAS}
    {{$.val:ALIAS}}
    {{ALIAS}}
    {{$.name:ALIAS}}
{end ALIAS}
```

- The previous example will write 3 times the same value or nothing if the values are not matching.

Path Block

- A path block is a block that is defined somewhere else it is actually inserted.
- A path block always start with a header such as `=== CLASS ===` where *CLASS* is the *java class* of the passed context (or the value of a `"class"` key, in case of a `Map`).
- The calling context class path may be added to form a sequence `=== CALLER/CLASS ===` so it can be referenced in the path block. This path is absolute and can only be called from the root context.
- Use the *any path* syntax `=== .../CLASS ===` to allow the path block to be called from any context of a given class.
- The path name **MUST NOT** contain spaces.

Calling

- To insert a path block, use the `call` operator inside a value `{{call PATH $.val}}`. Remember that the path **MUST** match the actual class or interface returned by the value expression.
- A path block can also be conditionnaly inserted `{{if $.test call PATH $.val}}`.
- It is possible to use alias to discriminate among multiple similar paths.
- Use the *any* `{{call .../PATH $.val}}` *path* notation to include any path blocks in the search.

```
{{call .../Class:Case1 $.valOfClass}} // Invoking a specific "case 1" for a Class

=== .../Class:Case1 ===
// Path block for case 1

=== .../Class:Case2 ===
// Path block for case 2
```

Referencing

- When inside a path block, you can access the current context value simply by its path name `{{Class}}` or it alias name `{{Case1}}`.
- The following example show how you can reference the calling block values.

```
=== .../CallingClass:ANCESTOR/MyClass:CURRENT ===

{{CallingClass.name}}.{{MyClass.name}} // Output both context names using their
path/class names
```

```
{{ANCESTOR.name}}.{{CURRENT.name}} // Output both context names using their path
aliases
```

JSON (not yet implemented)

- It is possible to use *JSON* files as root objects.
- *JSON* files are first loaded into `Map` objects.
- `Map` should have a `"class"` key to be used as a path.

```
// JSON expression emulating a Person object
// that will be loaded into a Map object
{class: "Person", name: "John"}
```

Call command (not yet implemented)

- When using a call command, it is possible to specify where you want to insert the text.
- Call and values are pretty similar except that call commands are defined with **single braces** `{call PATH VALUE}` just like controls.
- You can use conditional to decide when to invoke a command `{if !$.test call PATH $.val}`.
- It is not permitted to call the root context `{call $ $.val}` since this would trigger an infinite loop!

```
=== PATH ===
T0 // This is the normal text for this block
{call PATH_X $.val:T1} // PATH1 text is appended after this block (by default)
{call PATH_X $.val:T2 after} // Text is appended after the previous call
{call PATH_X $.val:T3 after PATH} // Text is appended after the previous call

// T0
// T1
// T2
// T3
```

- By default the `{call Class $.val}` command will be inserted after the current block. If the call command is invoked many times, then each invocation is *consecutive* to the preceding one.
- It is possible to insert the text *before* the current block. If the command is invoked many time, then each invocation is added *before* the previous one. It's just like the text has been glued before the current block and is now part of it.

```
=== PATH ===
T0 // This is the normal text for this block

{call PATH_X $.val:T1 before} // Text is appended before this block
{call PATH_X $.val:T2 before PATH} // Text is appended before the previous call

{call PATH_X $.val:T3} // Text is appended after the current block
```

```
{call PATH_X $.val:T4 after} // Text is appended after the previous call
{call PATH_X $.val:T5 after PATH} // Text is appended after the previous call

// T2
// T1
// T0
// T3
// T4
// T5
```

- It is possible to reverse the order by using the *right before* or *right after* commands.

```
=== PATH ===
T0 // This is the normal text for this block
{call PATH_X $.val:T1 right before} // Text is appended right before this block
{call PATH_X $.val:T2 right before PATH} // Text is also appended right before
this block

// T1
// T2
// T0
```

- In order to add some text *before* or *after* the whole document, use the special `$` path.

```
T0 // This is the normal text for this document
{call PATH $.val}
{call PATH_X $.val:T1 after} // Text is appended after the document
{call PATH_X $.val:T2 before} // Text is appended after the document

=== PATH ===
{call PATH_X $.val:T3 after $} // Text is appended after the document
{call PATH_X $.val:T4 before $} // Text is appended right before this block

// T2
// T4
// T0
// T3
// T1
```

- Where and when the `{call}` command is placed is as important as the command itself when determining *the placement* of the final text.

Comment, Escape & Ignore

- A comment begins with `%%%`. A comment terminates at the end of the line unless another `%%%` is encountered.

```
123%%% // Will output 123 without a new line
123%%%456%%% // Will output 123 WITH a new line (and this java comment!!!)
```

- The **escape** character ~ will make any immediately following character, including itself, to be treated as normal text. This is the only character that always needs to be escaped.
- Whenever possible, the engine will try to escape the invalid char sequences for you.

```
~{begin test~}%%% // Will output {begin test} (without this comment!)
```

- An empty line immediately before or after a path block header **is ignored**.
- When a line ends with a `{control}` the immediately following new line **is ignored**.

```
{begin ALIAS}
{{ALIAS}} // No new line added before the value!
```

- Notepad++ is a very handy companion when editing your template scripts. Simply double click on any alias to immediately visualize its structure.

Future developments

- Have a nice mecanism for error message handling.
- Should it use loggers or writers to dump the errors?
- Have a trace to follow the order of calling to debug the command calls
- Controls & commands in error are printed as is for convenience so it is easy to find the error in the script.
- Usage outside of Java

Please enjoy! S.Nadeau 😃