TopLeaf 7 How To:

Using Stacked Variables

1. Introduction

Stacks are useful programming constructs that can be used to maintain context or state information. Data pushed onto a stack can be used to indicate the current value of a variable within a nested context. TopLeaf's variable stacks are available through the Map Manager's custom boxes. One application of stacks in TopLeaf is nested lists. Another is maintaining a variable built from hierarhical data.

Stacking Things Up

A stack is a programming construct which allows data to be stored by pushing new values on top of the stack. Many programming languages have special functions, usually named "push" and "pop", for putting data on the stack, and removing data from the stack, respectively.

If you are unfamiliar with programming with stacks, think of a stack like any other stack: a stack of books, magazines, newspapers – any "pile". You normally add items to these piles by putting new items on top. Usually, the only reliable way to get things out of the pile is to remove things from the top until you reach the item you want.

A programming stack works the same way. The item placed on the stack stays on the stack until it is removed. When it is removed, the previously stacked item becomes accessible.

TopLeaf uses stacks extensively to maintain mapping context information. In fact, each mapping's information is pushed onto or removed from a stack whenever a tag is encountered in the data stream. Consequently, the only information available at any point in processing a document is the information currently on the stack(s).

TopLeaf provides you with the ability to stack variables in custom boxes – both for regular and custom tag mappings. A common use of these kinds of stacks is for list numbering.

Normally, you create numbered lists by defining a tag mapping as a one of the numeric list item types. There are many differnt options to choose from: you can create lists that are decimals, upper-case alphas, lower-case alphas, upper-case roman numerals, or lower-case roman numerals. Problems start to arise when you have numeric nested lists that go deeper than the five distinct levels available, because there are limitations on list items when applying formatting to the list labels.

List labels can be modified in the **$document** mapping using the **<listitem-properties/>** command. List item label formats are not stacked, however, so you cannot change the format depending on which level within a nested list you are in.

Consider an application that has seven levels of nesting:

```
A. A First-Level List Item
  (1) A Second-Level List Item
    (a) A Third-Level List Item
      [1] A Fourth-Level List Item
        [a] A Fifth-Level List Item
          [i] A Sixth-Level List Item
            [A] A Seventh-Level List Item
```

Note that both the first and seventh levels are upper-case alpha, and the second and fourth are decimal. But they have different label formats: parentheses are used in some cases, square brackets in others.

There is no way to implement the different label formats required, using the built-in list item mappings.

The only way to do this is to implement the lists using variable stacks.

Mapping the Structure

Let's begin by supposing our nested list structure looks something like this:

```
<level>
 <litem>A First-Level List Item</litem>
  <level>
  <litem>A Second-Level List Item</litem>
    <level>
    <litem>A Third-Level List Item</litem>
      <level>
      <litem>A Fourth-Level List Item</litem>
        <level>
        <litem>A Fifth-Level List Item</litem>
          <level>
          <litem>A Sixth-Level List Item</litem>
            <level>
            <litem>A Seventh-Level List Item</litem>
            </level>
          </level>
        </level>
      </level>
    </level>
  </level>
</level>
```

There are only two elements involved in this nested structure: "level" and "litem". At any of the seven levels, you might have dozens of list items. For any given level, list item numbers restart at 1. For the items within any one level, the numbers increment.

In handling this, there are two approaches we could take. First, we can provide a single mapping for **level/litem**, and use counters to determine which label format we want to implement based on the value of a counter. Second. we can provide distinct mappings for each level, and call a different custom marker for each label, for each **level/litem** mapping in its respective nested context.

In the first case, we can use stacked variables to determine which level's label we are to use. In the second case, we don't need a stacked variable for this, but we can just use the stacked variable to initialize the starting value for the list.

Let's start with the second way first, that is, by providing a distinct mapping for each nested level.

Creating the Mappings

We will need to create mappings for each level in context. To do this, we run the Map Manager, and create tag mappings in context for:

```
level/litem
level/litem/level/litem
level/litem/level/litem/level/litem
level/litem/level/litem/level/litem/level/litem
level/litem/level/litem/level/litem/level/litem/level/litem
level/litem/level/litem/level/litem/level/litem/level/litem/level/litem
level/litem/level/litem/level/litem/level/litem/level/litem/level/litem/level
/litem
```

The mapping names are quite long, and so a bit difficult to read in the map manager window.

Next, in the custom box for each, we add this line:

```
<stack var="ListNumber" value="{tag-occurrence}"/>
```

There are certain constraints on using stacks. The main one is that to push a value onto a stack, the **<stack/>** declaration must be the first thing in the custom box for the mapping. If it occurs anywhere else, errors will get generated during document composition. If you are stacking multiple variables, the <stack/> declarations must follow one another, with nothing else between them. You cannot stack the same variable more than once within any one mapping.

The other thing to remember about stacks is that they cannot be "popped" explicitly. Technically, the **<stack/>** command allows local copies of variables to be placed on the current mapping stack. Each value pushed onto a TopLeaf variable stack automatically pops off the stack when the current mapping completes, because the mapping itself and all information associated with it is removed from the stack.

So, in this case, the "ListNumber" variable will be set to "{tag-occurence}", which translates to the index of the **litem** tag being processed at the time. This becomes a local variable available to the mapping, and any custom markers called from the mapping.

We need to define a custom marker for each level of mapping. You can either create the custom marker first, and then update the custom boxes for each of the "litem" mappings, or you can add the custom marker reference to the custom boxes first, and then define them after the fact.

My personal preference is to add the reference to the custom box first.

The custom boxes for each of the mappings ends up like this:

level/litem

```
<stack var="ListNumber" value="{tag-occurrence}"/>
<Label1>
```

level/litem/level/litem

```
<stack var="ListNumber" value="{tag-occurrence}"/>
<Label2>
```

level/litem/level/litem/level/litem

```
<stack var="ListNumber" value="{tag-occurrence}"/>
<Label3>
```

level/litemlevel/litem/level/litem/level/litem

```
<stack var="ListNumber" value="{tag-occurrence}"/>
<Label4>
```

level/litem/level/litem/level/litem/level/litem/level/litem

```
<stack var="ListNumber" value="{tag-occurrence}"/>
<Label5>
```

level/litem/level/litem/level/litem/level/litem/level/litem/level/litem

```
<stack var="ListNumber" value="{tag-occurrence}"/>
<Label6>
```

level/litem/level/litem/level/litem/level/litem/level/litem/level/litem/level/litem

```
<stack var="ListNumber" value="{tag-occurrence}"/>
<Label7>
```

Each of the custom "Label" markers ("Label1" through "Label7") is created as a "Label" category of paragraph. This defaults an offset to the labels, so that they hang to the left of the paragraph being set. Make sure the the left margin setting of the "level/litem" mappings is at least equal to the value of the offset for each of the label custom markers, otherwise, you will get compositional errors (attempting to move past the left margin).

In the custom box for each of the label markers, add the pre-content to reference the "{ListNumber}" variable, and set the formats as required. For some mappings in this example, this means transforming the value of {ListNumber} to an alpha or roman numeral representation.

Label1 custom content:

```
<set var="Label" value="{ListNumber}" transform="ALPHA,UPPER"/>
{Label}.
```

Label2 custom content:

```
<set var="Label" value="{ListNumber}"/>
({Label})
```

Label3 custom content:

```
<set var="Label" value="{ListNumber}" transform="ALPHA,LOWER"/>
({Label})
```

Label4 custom content:

```
<set var="Label" value="{ListNumber}"/>
[{Label}]
```

Label5 custom content:

```
<set var="Label" value="{ListNumber}" transform="ALPHA,LOWER"/>
[{Label}]
```

Label6 custom content:

```
<set var="Label" value="{ListNumber}" transform="ROMAN,LOWER"/>
[{Label}]
```

Label7 custom content:

```
<set var="Label" value="{ListNumber}" transform="ALPHA,UPPER"/>
[{Label}]
```

Because "ListNumber" is stacked, its actual value will vary depending on when it is evaluated. Each time an "litem" mapping is encountered, a new value of "ListNumber" is pushed onto the stack. When the "litem" mapping completes, the current value of "ListNumber" on the stack is removed from the stack. This way, you can traverse through linear an nested levels of list items without any great complication of interacting counters.

Each of the label custom markers outputs a transformed representation of "ListNumber", with the appropriate parentheses or trailing dots. By creating a mapping for each level of "litem", and using stacked variables, we have created a fully functional 7-level nested list that is very easy to maintain.

Multiple Stacks

The multiple mappings in the previous implementation might get unwieldy, so you might start looking for alternatives.

In this example, we combine a flat mapping structure with two stacked variables: one for "level", and one for "litem".

Let's start by deleting all the "level/litem" mappings. Next, we create a mapping for "level", and one for "litem". We're going to use a variable to count which level we are in, so the first thing to do is to initialize it in the custom box for the **$document** pseudo-tag.

```
<set var="Level" value="0"/>
```

With that done, we can go to our "level" mapping and set up its pre-content custom box like this:

```
<stack var="Level" value="{Level}+1"/>
```

The "litem" mapping's pre-content custom box will contain:

```
<stack var="ListNumber" value="{tag-occurrence}"/>
<switch>
<case var="Level" target="1">
<Label1>
</case>
```

```
<case var="Level" target="2">
<Label2>
</case>
<case var="Level" target="3">
<Label3>
</case>
<case var="Level" target="4">
<Label4>
</case>
<case var="Level" target="5">
<Label5>
</case>
<case var="Level" target="6">
<Label6>
</case>
<case var="Level" target="7">
<Label7>
</case>
</switch>
```

As each nested "level" is encountered, the local copy of the "Level" counter is incremented by one from the previous local copy's value. As each "level" terminates, the counter is decremented. Using stacks, we don't have to worry about decrementing the counter inappropriately.

Interestingly, you cannot have a flat mapping structure like this and use a stacked variable on "level" mappings based on the value of "{tag-occurrence}", because each list level nested within a "level" mapping will always initialize with a "{tag-occurence}" of 1. This means that regardless of how levels are nested, all levels will be composed as if they are top-level lists.

This approach improves on the original implementation by providing more flexibility in the tag mappings, at the expense of adding a bit of complexity in the "litem" custom box, by having a seven-layer switch.

Context Stacks

Another application where stacks come in handy is in situations where you need to manage the context within a document. In some documents, you may need to buld a "path" of hierarchical information based on sub-sections or sub-components in a document, such as you might find in a bill of materials, or other hierarchical information. You can use stacks to dynamically add to and remove from the overall path.

Let's take a concrete example. Suppose you have a document structure that contains:

```
<Component name="Vaccuum Cleaner">
 <Component name="Hose">
  <Component name="Flange Lock"></Component>
  <Component name="Flex-Hose"></Component>
  <Component name="Nozzle Mount"></Component>
  <Component name="Nozzle"></Component>
 </Component>
 <Component name="Cannister">
  <Component name="Power Switch"></Component>
  <Component name="Electric Cord"></Component>
  <Component name="Power Connector"></Component>
  <Component name="Cover">
   <Component name="Cover Latch"></Component>
   <Component name="Seal"></Component>
   <Component name="Hinge"></Component>
  </Component>
 </Component>
</Component>
```

Add an initializer for a "Path" variable in "$document":

```
<set var="Path" string=""/>
```

In the pre-content custom box for the "Component" mapping, add:

```
<stack var="Path" string="{Path}{@name}/"/>
{Path}
```

When composed, the output looks like:

```
Vaccuum Cleaner/
Vaccuum Cleaner/Hose/
Vaccuum Cleaner/Hose/Flange Lock/
Vaccuum Cleaner/Hose/Flex Hose/
Vaccuum Cleaner/Hose/Nozzle Mount/
Vaccuum Cleaner/Hose/Nozzle/
Vaccuum Cleaner/Cannister/
Vaccuum Cleaner/Cannister/Power Switch/
Vaccuum Cleaner/Cannister/Electric Cord/
Vaccuum Cleaner/Cannister/Power Connector/
Vaccuum Cleaner/Cannister/Cover/
Vaccuum Cleaner/Cannister/Cover/Cover Latch/
Vaccuum Cleaner/Cannister/Cover/Seal/
Vaccuum Cleaner/Cannister/Cover/Hinge/
```

3. Conclusion

Stacking variables is a convenient way to handle recursive counters, especially in such things as nested lists. They can also be used to maintain hierarchical paths or other context information, without the complications necessary when attempting to do these tasks with counters.

About the Author

John Barker is the co-founder and president of Metaformix Information Systems Inc. He has developed integrated publishing solutions around TopLeaf since 1998.

Contact him at john[_at_]metaformixis.com