StaxMateIterators

StaxMate Cursors

The input side of StaxMate is built on concept of multiple synchronized **cursors**. Cursors can be thought of as pointers that can only move forward over a scope: a subset of the document they can traverse over. Scope is either a sub-tree for a given element (including all of its ancestors, children and grand-children), or a "slice" that contains just immediate children. That is, a cursor can only traverse the sub-tree it was created for.
During traversal, you can create new child cursors for existing cursors.

The only limitation regarding cursor usage is that only one of cursors is ever active; that is, points to valid content to be accessed.
A cursor is automatically activated when it is created, or when it is advanced. Cursor gets completely deactivated when it traverses through its scope: this also happens for all child cursors when the parent cursor is advanced (because it has to skip the sub-tree(s) its child cursors would traverse).

If this sounds confusing, please refer to Tutorial page, and sample usage: sample code should be very intuitive and show the reasons for above-mentioned restrictions.

Types of Cursors

All cursors extend **SMInputCursor** base class; and most API is defined at this class. But there are 2 main flavours of cursors, implemented as sub-classes:

- **nested cursors** (or "child cursors")
  - implemented by SMNestedCursor
  - traverse only the immediate children of the parent element
  - only expose START_ELEMENT events matching each immediate child element (end of the scope is implicitly known via getNext() return 'null')
  - is the usual cursor used when doing hierarchic (recursive-descendant) parsing or transformations.
- **flat cursors** (or "descendant cursors")
  - implemented by SMFlatCursors
  - traverse full sub-trees presenting "flattened" view (and exposing both START_ELEMENT and END_ELEMENT events; latter is needed due to flattening, to know nesting boundaries)
  - Useful when, for example, collecting all text in a sub-tree; or when trying to search for a specific element regardless of its relative location within tree

Creation of cursors is done either via **SMInputFactory** (for root cursors), or via other cursors (for all child and descendant cursors): each time an cursor points to START_ELEMENT, a new child cursor (flat or nested) can be created.

Why Cursors?

The main benefit of cursors is scoped access but serialized (document-order) access to XML content – this is required to be able to use basic Stax XMLStreamReader without additional buffering.
What this means is that:

#. It is always safe to pass a child cursor to another processing component: that component can only access entries within scope of the cursor; and all this without that component having to keep track of the nesting of elements.
#. Access via multiple dependant cursors (child, parent) is serialized such that underlying access is always in document order.

Point 2. means that when a component is done with cursor (at any point; including not using it at all), there is no need to manually "fast forward" through events that cursor would be seeing. Parent cursor takes care of skipping through events not needed, automatically, when parent cursor itself is advanced. This advancement will then invalidate all child cursors (since cursor has advanced past point where they would be valid), and ensuring they can not be advanced any more.

Access to event information

All normal access to the event information (name of the element, attributes, attribute values, textual content, processing instruction target and so on) is accessible via the active cursor (one that is currently pointing to an underlying Stax event). At any given point, there will be just one such cursor. Although there are methods to check if an cursor is at such valid point, this is seldom needed: as long as access is done right after advancing a cursor, it **"just works"**.

Filtering cursors

In addition to the two main types of cursors, there is also support for simple configurable filtering of events visible using the cursors. For example, it is trivially easy to construct an element-only cursor (one that ignores all other event types but START_ELEMENT (and for flat cursors, END_ELEMENT)):

SMInputCursor nestedTextIter = currentCursor.childElementCursor();
SMInputCursor nestedTextIter = currentCursor.descendantElementCursor();

(first call creates a nested cursor, and second a flat cursor)
And for more configurable filtering:

SMInputCursor nestedTextIter = currentCursor.childCursor(new SimpleFilter(...));
SMInputCursor nestedTextIter = currentCursor.descendantCursor(new SimpleFilter(...));

you can specify your own filtering rules (SimpleFilter strictly bases filtering on event types; simple, fast and usually sufficient – it's used to implement text and element filters).

Tracking

Tracking is a simple yet powerful mechanism for persisting some subset of information for the currently active branch of the logical XML content tree. For example, you may want to know element and attribute names, and all attribute values, of all the parents of the element an cursor currently points to; but you need not keep track of any other ancestor information. This may be decent compromise between full in-memory DOM and transient streaming processing.

Tracking can be enabled on per-cursor basis: and it takes effect for all child (and descendant) cursors of the cursor. This is because there is no way to retrieve information of events that have been passed already.

(**to be completed**)

Typed Access

(new with StaxMate 2.0 – **to be complete**)

Convenience methods

There are some additional convenience methods, for doing commonly needed thins like:

- Get me all the text contained within the element this cursor points to (ignoring all elements, comments, processing instructions, if any)

(**to be completed**)