Introducing The Janus XML Parser

John Thickstun
Sirius Software Inc.

Sirius Software, Inc.

XML Parsing

- XML is defined as a character stream
  - But it describes hierarchical data
- Sirius already has the XmlDoc API to convert XML character stream into an internal tree structure
  - Similar to MS/XML, Java DOM, etc.
  - Allows easy and efficient navigation of data hierarchy using method calls
  - API also converts back from tree structure to XML
  - The entire XML document is always parsed by the XmlDoc API

The Janus XmlParser Class

- A new class for doing XML parsing
- A stream/event oriented API
  - User Language application is notified when potentially interesting parts of the XML document are encountered
- Does **not** convert the entire XML document to any structured object
- Can even only parse the start of an XML document

So Why Are We Interested in an Event-Based XML Parser for User Language?

- Because sometimes one is only interested in a small part of an XML document
  - So want to avoid the overhead of parsing and structuring the entire document
- Because it illustrates some interesting aspects of recent Janus SOAP user Language Enhancements
  - And illustrates how Janus SOAP User Language Interface is at the forefront of modern programming language developments
- But keep in mind: for **most** applications it is better to use the XmlDoc API to process XML documents

SAX, the Industry Standard

- Simple API for XML
- More of a recommended framework than a standard
  - The Java implementation of SAX is considered to be normative
    - http://en.wikipedia.org/wiki/Simple_API_for_XML
- Event-based
  - When parser hits a certain XML entity, it calls an application-specific handler for that entity
    - The start and end of an XML element being the most commonly used
- Inheritance based
  - Callbacks are specified via methods that implement an abstract class

The Problems With SAX

- It's the application's job to filter interesting elements from uninteresting elements
  - Inefficiency – a call for every element, even if uninteresting
- It's the application's job to keep track of element context
  - So application might need to maintain its own stack
- Both problems fall out of the fact that SAX callbacks are defined using inheritance
  - So can't have element/context-specific handlers
  - Because only one method can override another in a single class
  - Inheritance is also a bit of a heavyweight approach to doing XML parsing

The Janus XML Parser API

- Event-based like SAX
- But does not use inheritance for specifying event handlers
  - Instead uses method variables
    - This option was not available in SAX because many languages (including Java) don't support method variables
  - So context and element-specific handlers supported
- Handlers are only set for the elements the user cares about

XML For Examples

```
<collection>
   <artist name="Radiohead">
      <album name="The Bends">
         <song length="4:51" title="Fake Plastic Trees">
            Her green plastic watering can
         </song>
         <song length="3:43" title="Sulk">
            You bite through the big wall, the big wall bites back
         </song>
      </album>
      <album name="OK Computer">
         <song length="4:44" title="Airbag">
            In the next world war
         </song>
         <song length="6:24" title="Paranoid Android">
            Please could you stop the noise, I'm trying to get some rest
         </song>
         <song length="4:22" title="Karma Police">
            Karma police, arrest this man
         </song>
      </album>
   </artist>
</collection>
```

Listing Names and Titles from the XML Document

```
%xml    is longstring
%parser is object xmlParser
...
local function (xmlParser):startSong(%name is unicode namerequired, -
                          %attrlist is object xmlAttributeList) -
                          is object xmlSelector
   printText title = {%attrlist:value('title')}, length = {%attrlist:value('length')}
   return null
end function

%parser = new
%parser:string = %xml
%parser:defaultSelector = startElement('song', startSong)
%parser:parse
...
```

```
title = Fake Plastic Trees, length = 4:51
title = Sulk, length = 3:43
title = Airbag, length = 4:44
title = Paranoid Android, length = 6:24
title = Karma Police, length = 4:22
```

That's Not Very Interesting

- No data being returned to application that tells the parser to do the parse

- The solution is to use an Expose[d] local function to share data with the calling/containing code
  - This will be very common for XmlParser applications

Extracting Names and Titles from the XML Document

```
%xml        is longstring
%parser     is object xmlParser
%songLength is namedArraylist of string len 8
...
local function (xmlParser):startSong(%name is unicode namerequired, -
                          %attrlist is object xmlAttributeList) -
                          is object xmlSelector expose
   %songLength(%attrlist:value('title')) = %attrlist:value('length')
   return null
end function
...
%parser = new
%parser:string = %xml
%parser:defaultSelector = startElement('song', startSong)
%songLength = new
%parser:parse
for %i from 1 to %songLength:count
   printText Length of "{%songLength:nameByNumber(%i)}” = ...
   printText {%songLength:itemByNumber(%i)}
end for
...
```

```
Length of "Airbag” = 4:44
Length of "Fake Plastic Trees” = 4:51
Length of "Karma Police” = 4:22
Length of "Paranoid Android” = 6:24
Length of "Sulk” = 3:43
```

That's a Little Tacky

- We use the DefaultSelector method to avoid worrying about the XML document hierarchy
- But this means we could accidentally pick up a <song> element in the wrong context
- Also slightly less efficient because parser must do checks for every element
- So let's see how the code looks if done “properly”
  - Note in the following code that we match on any outer element name

Extracting Names and Titles from the XML Document

```
local function (xmlParser):startSong(%name is unicode namerequired, -
                          %attrlist is object xmlAttributeList) -
                          is object xmlSelector expose
   %songLength(%attrlist:value('title')) = %attrlist:value('length')
   return null
end function

local function (xmlParser):startAlbum is object xmlSelector expose
   return startElement('song', startSong)
end function

local function (xmlParser):startArtist is object xmlSelector expose
   return startElement('album', startAlbum)
end function

local function (xmlParser):startOuter is object xmlSelector expose
   return startElement('artist', startArtist)
end function
...
%parser = new
%parser:string = %xml
%songLength = new
%parser:parse(startElement('collection', startOuter))
... (same as previous example)
```

But That's Still a Little Tacky

- The relationship of the StartElement methods is unclear
- If there are several elements being selected, things can get really messy
- So let's see a nicer way of doing this

Extracting Names and Titles from the XML Document

```
%path           is arraylist of unicode
%path = list('collection', 'artist', 'album', 'song')

local function (xmlParser):startPush -
                           (%attrlist is object XmlAttributeList nameRequired) -
                           is object xmlSelector expose
    if %this:depth eq %path:count then
        %songLength(%attrlist:value('title')) = %attrlist:value('length')
        return null
    end if

    return startElement(%path(%this:depth + 1), startPush)
end function

%parser = new
%parser:string = %xml
%songLength = new
%parser:parse(startElement('collection', startOuter))
... (same as previous example)
```

Suppose You Want to Extract Element Text (First Line of Song)

- EndElement selector required instead of StartElement
  - But only for the elements whose text is required
  - Generally this would be only be for innermost elements
  - Avoid EndElement for elements whose values you're not interested in
    - Can significantly increase overhead

Listing First Lines of Songs from XML Document

```
...
local function (xmlParser):startOuter is object xmlSelector expose
   local function (xmlParser):startArtist is object xmlSelector expose
      local function (xmlParser):startAlbum is object xmlSelector expose
         local subroutine (xmlParser):endSong(%name is unicode namerequired, -
                                          %text is unicode) expose
            print %text:unspace(spaces='4025':x)
         end subroutine
         return endElement('song', endSong)
      end function
      return startElement('album', startAlbum)
   end function
   return startElement('artist', startArtist)
end function

%parser = new
%parser:string = %xml
%songLength = new
%parser:parse(startElement('*', startOuter))
%parser:parse
...
```

```
Her green plastic watering can
You bite through the big wall, the big wall bites back
In the next world war
Please could you stop the noise, I'm trying to get some rest
In the next world war Please could you stop the noise, I'm trying to get some rest
Karma police, arrest this man
```

That's Not Likely to Be Very Useful

- Probably want to associate first line of song with title
- But title is an attribute and first line is element text
  - So title only available in StartElement handler
  - And first line only available in EndElement handler
  - So need some place to store the title (or attribute list) between StartElement handler and EndElement handler
    - A *Shared* variable is probably the best solution
    - Mumble, mumble, closures, mumble, mumble

Extracting First Lines of Songs by Title from XML Document

```
...
local function (xmlParser):startOuter is object xmlSelector expose
   local function (xmlParser):startArtist is object xmlSelector expose
      local function (xmlParser):startAlbum is object xmlSelector expose
         %title is unicode shared
         local function (xmlParser):startSong(%name is unicode namerequired, -
                                  %attrlist is object xmlAttributeList) -
                                  is object xmlSelector expose
            %title = %attrlist:value('title')
            return null
         end function
         local subroutine (xmlParser):endSong(%name is unicode namerequired, -
                                  %text is unicode) expose
            %songFirstLine(%title) = %text:unspace(spaces='4025':x)
         end subroutine
         return list(startElement('song', startSong), endElement('song', endSong))
      end function
      return startElement('album', startAlbum)
   end function
   return startElement('artist', startArtist)
end function
...
%parser = new
%parser:string = %xml
%songFirstLine = new
%parser:parse(startElement('*', startOuter))
...
```

Taking a Deep Breath

- Yes, there are some fairly deep concepts at play in the XmlParser API
  - SAX revolving around inheritance has similarly complex issues
  - Don't worry if you don't completely “get it” right away
- You can use this presentation (and eventually the doc) as a pattern for your own applications
- Probably most people won't need to do event-based XML parsing anyway
  - But worthwhile learning the concepts anyway in case they apply to some other problem
- Let's quickly go over the pieces

The XmlParser Class

- Drives the XML parsing process
  - Current parsing state held inside class
  - Responsible for calling the event handlers
- Contains a unicode string with the XML
  - Yes, the whole process is unicode
  - Beware conversion issues, but usually there aren't any
- Parse started with the Parse method
  - Always zero or one XmlSelector active
  - If no XmlSelector active, the XML is scanned for validity but nothing else is done with it
  - Current active XmlSelector pushed/pop at start/end of element tag
- StopParsing method will stop parsing

The XmlSelector Class

- Used to tell the XmlParser the XML bits of interest and the event handlers for those bits
- Objects are immutable (read-only after creation)
- So consist mostly of constructors
  - StartElement – For element start tag
    - Name of element and handler are arguments
  - EndElement – For element end tag
    - Name of element and handler are arguments
  - ProcessingInstruction
    - Name of PI and handler are arguments
  - List
    - One or more XmlSelector objects are arguments

XmlAttributeList Class

- Contains list of attributes passed to StartElement handler
  - More or less like a NamedArrayList
  - But has namespace support so two indexes to each item
    - Default namespace is none
- Attribute values can be retrieved by name
  - The Value method
- Attribute names and values can be looped over
  - Using the Count, LocalName, and ValueByNumber methods
  - Also a bunch of methods for dealing with namespaces
    - Won't bore you with them here

XmlStartHandler, XmlEndHandler, and XmlPIHandler Types

- Implicit type of handler name in XmlSelector constructor
- But can declare variables of this type
  - %foo is type XmlStartHandler
  - For variables in XmlSelector constructor

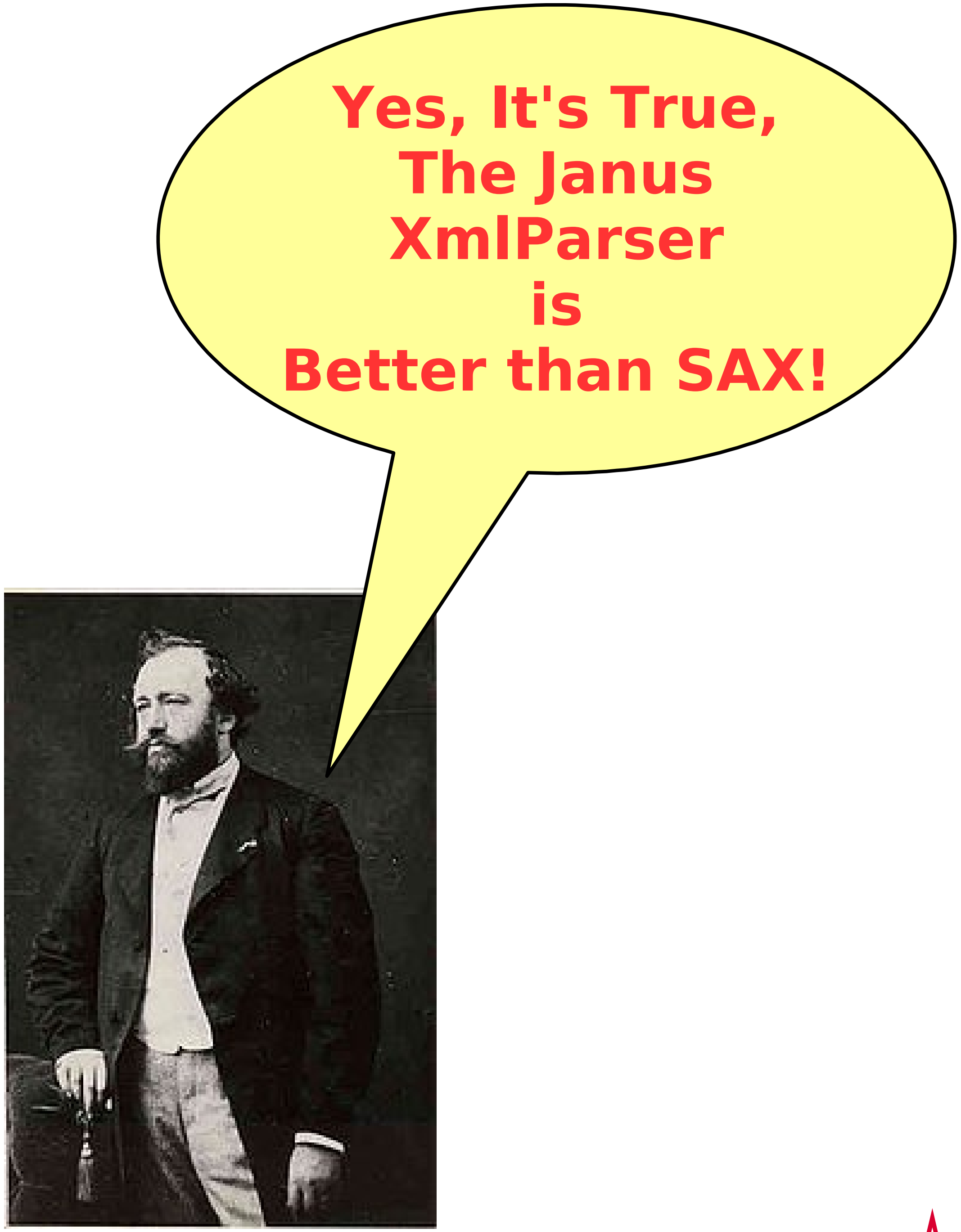
    ```
    return startElement('foo', %foo)
    ```
  - `Also:`
    - `%foo is type XmlEndHandler`
    - `%foo is type XmlPIHandler`

Conclusions

- XmlParser API provides a new, powerful way of extracting a few bits of info out of an XML document
- The XmlParser API is based on a fairly small set of classes and methods
- But is based on a fairly deep set of concepts
  - Well worth learning the concepts even if you never need the API
  - Many of the same concepts apply to managing collections
  - These concepts will probably appear more often as we figure out more useful ways to apply them
- Much more powerful than what poor Java programmers have available to them (SAX)
  - More efficient
  - More convenient to code

Questions?

Comments?



Sirius Software, Inc.