



# **XML::SAX - an introduction**

**Dominic Mitchell**

# XML::SAX - what is it?

- Modern XML Processing for Perl
- Stream based parsing
- Object Oriented interface
- Parser Agnostic
- Pipelines
- Written by Matt Sergeant / Kip Hampton / Robin Berjon
  - ◆ Based on discussion on the perl-xml list
  - ◆ And experiences from PerlSAX
- Generally, a good idea

# XML::SAX - what is it?

- Modern XML Processing for Perl
- Stream based parsing
- Object Oriented interface
- Parser Agnostic
- Pipelines
- Written by Matt Sergeant / Kip Hampton / Robin Berjon
  - ◆ Based on discussion on the perl-xml list
  - ◆ And experiences from PerlSAX
- Generally, a good idea
  - ◆ Even though it was nicked from Java

# XML::SAX - why should I care?

- Standard
  - ◆ Reasonably similiar to SAX for Java, Python, etc.
- Portable
  - ◆ Will work on *any* perl installation.
  - ◆ No XML::Parser install required!
- Low Memory Consumption
  - ◆ Streaming means only one tag at a time in memory
  - ◆ Can be quicker than non streaming parser
- Pipelines
  - ◆ Plug different tasks together easily

# XML::SAX - the bad bits

- It's XML
  - ◆ Reason enough for many people
- Streaming Parser
  - ◆ No lookahead or behind during parse
  - ◆ This is often excessively restrictive
- Immature tools
  - ◆ Still a few bugs around the edges
    - XML::SAX::PurePerl took up lots of memory for me
    - XML::SAX::Expat doesn't supply a locator
    - XML::LibXML::SAX isn't 100% on namespaces
  - ◆ Although generally not critical

# XML::SAX - how does it work?

- A method is called on your object for each event

```
package Foo;
use base 'XML::SAX::Base';
sub start_element {
    my $self = shift;
    my ($data) = @_;
    print "I saw a $data->{Name} element!\n";
    $self->SUPER::start_element( $data );
}
```

- Standard method names based on events
  - ◆ See XML::SAX::Base for a complete list
- Hash ref gets passed in with event details
- Must pass on event
- Your class must inherit from XML::SAX::Base

# XML::SAX - how do I use it?

```
use XML::SAX::ParserFactory; use Foo;
my $handler = Foo->new();
my $parser = XML::SAX::ParserFactory->parser(
    Handler => $handler
);
$parser->parse_string( '<foo>bar</foo>' );
```

- Instantiate your object
- ParserFactory finds an appropriate parser
  - ◆ Although it can be guided...
- The parser gets given your object
- Events will be passed to your object during the parse

# XML::SAX - Pipelines

- The code just shown is a *SAX filter*
- These filters can be plugged together

```
use XML::SAX::ParserFactory; use Foo; use Bar;
my $handler = Bar->new();
my $handler2 = Foo->new( Handler => $handler );
my $parser = XML::SAX::ParserFactory->parser(
    Handler => $handler2
);
$parser->parse_string( '<foo>bar</foo>' );
```

- Handlers passed in to each others constructor
- In *reverse* order
- There is an easier way (more later)

# XML::SAX - without the xml!

- You don't have to use an XML parser
- So long as you call the right methods...
- And your object implements parse\*()
  - ◆ Known as a Generator (or Producer in Java)
- Very handy for migrating other data sources to XML
- And using XML tools with non XML data.
- eg: XML::Generator::Directory
  - ◆ Turns opendir() into SAX events

# XML::SAX - useful addons

- XML::SAX::Writer
  - ◆ Outputs your XML *correctly!*
- XML::SAX::Machines
  - ◆ Makes managing pipelines much easier
- XML::Filter::BufferText
  - ◆ Makes all character data arrive together
- XML::Handler::AxPoint
  - ◆ For writing this presentation
- And many more...
  - ◆ Search for XML::Filter and XML::Generator on CPAN
- saxdump.pl
  - ◆ Good example to write yourself and see what goes through
  - ◆ Data::Dumper + AUTOLOAD

# XML::SAX::Machines

- Much easier way to plug filters together

```
use XML::Sax::Machines qw( Pipeline );  
my $machine = Pipeline( Foo => Bar => \*STDOUT );  
$machine->parse_string( '<foo>bar</foo>' );
```

- Lots of other ways for combining SAX filters
- Pipeline() is the most common

# XML::SAX::Nibbler



- Nibbler says: use XML::SAX to save the universe!