Advanced Vim Syntax Programming and Scripting

Announcement

Managements requests that all the Emacs chauvinist in the audience refrain from arguing with the Vim chauvinist on stage concerning which editor is better.

Such arguments are generally considered intellectual combat between two unarmed opponents.

Topics

- Programming the Syntax Engine
- Creating keyboard macros
- Using the :map command
- Basic Functions
- Creating a simple function
- Connecting the editor and the function
- Combining syntax coloring and function definitions.

Topics

- Perl Programming
- Advanced Features

The Instructor

Steve Oualline

- Author of “Vim (Vi Improved)”
- Contributed tutorial text material to the Vim project
- Website http://www.oualline.com
- E-Mail: oualline@www.oualline.com

Cheat Sheets

- You should all have a copy of the cheat sheets showing the scripts and examples being discussed here.
- These will be referenced throughout the tutorial.

- Cheat sheets and slides can be downloaded from:

http://www.oualline.com/vim.talk

Programming the Syntax Engine

Cheat Sheet Time

- We will be discussing

  1) syntax.sl – Sample file written in a strange language.

  2) syntax.vim – Syntax coloring rules for “sl”

Getting Started

Clear out the old syntax

```
:syntax clear
```

Tell Vim if the language is case sensitive

```
:syntax case ignore
:syntax case match
```

Highlight

- To see the names of the various highlight groups

```
:highlight
```

- To define your own

```
:highlight StrangeWord
\ guifg=purple
\ guibg=yellow
```

- (Many more options available)

Define Some Keywords

To define a keyword

```
:syntax keyword LangWord if then
```

Use a different highlight for system functions.

```
:syntax keyword Function print
```

Define elements that match a regular expression

Define a match for an identifier

```
:syntax match Identifier
\               /[_a-zA-Z][_a-zA-Z0-9]*/
```

Now define an element that matches numbers.

```
:syntax match Number /[0-9]\+/
```

Defining a region

Comments start with “comment”

End with “end-comment”

Vim syntax definition is:

```
:syntax region Comment
\     start=/comment/
\     end=/end-comment/
```

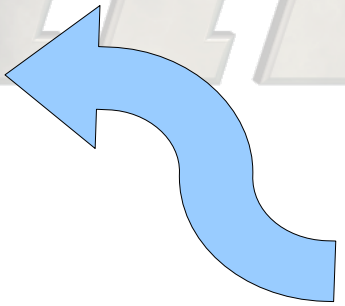
Defining a string

The problem:

```
“String with \” in it”
```

The solution

```
:syntax region String
\     start=/"/ end=/"/
\     skip=/\\"/
```



Highlighting TODO in comments

Define the region to be highlighted, but only if “contained” in another element

```
:syntax region Todo \
\    start=/TODO:/ end=/$/
\    contained
```

Highlighting TODO in comments

Tell Vim that a comment can contain a TODO:

```
:syntax region Comment
\    start=/comment/
\    end=/end-comment/
\    contains=Todo
```

Defining a one line syntax element

Normally matches span lines. But with the "oneline" option they do not.

```
:syntax region PreProc
\    start=/^#/
\    end=/$/ oneline
```

Continuation Lines

Revised PreProc – Notice contains clause.

```
:syntax region PreProc
\       start=/^#/ end=/$/
\       oneline
\       contains=LineContinue
```

- Note: You must clear out the old PreProc definition before using this:

```
:syntax clear PreProc
```

Continuation Lines

Define continuation line

```
:syntax match LineContinue
\              /\\\n.*/ contained
```

Since this *can* contain a newline it continues the previous line.

Fixing the highlight the same

Highlight both syntax elements the same.

```
:highlight link
\       LineContinue PreProc
```

Extreme Syntax Coloring

- Rainbow.vim – Syntax coloring to highlight nested parenthesis.
- Use the option matchgroup to indicate the group ends match one highlight and the body another.

Autoloading Syntax Files

- Local Syntax files go in

  $VIMRUNTIME/syntax/<l>.vim

Where <l> is the language name (as defined by the filetype option)

Note: Filetype is controlled by the file:

$VIMRUNTIME/filetype.vim

Syntax Help

- To get information about how to write a syntax file use the command:

  `:help :syntax`

- This file also describes the language specific options. For example the C specific options **`c_gnu`**

C Specific Options

- c_gnu

  GNU gcc specific items

- c_comment_strings

  strings and numbers inside a comment

- c_space_errors

  trailing white space and spaces before a <Tab>

Advanced Syntax Items

- Any item which contains @Spell will be spell checked.
- Any item which contains @NoSpell will not be spell checked.

Keyboard Macros and Mapping

Cheat Sheet Time

- Follow along in

  `i-map.vim`

Keyboard Macro Example

The problem change lines like:

```
foo.h
```

into

```
#include <foo.h>
```

for lots of lines.

Keyboard Macros

q*{register}* Record commands into a register

{commands} Execute commands

q End recording

@*{register}* Execute keyboard macro

Keyboard Macro Example

qa – Start recording in register a

^ – Go to beginning of line

i#include < *esc* – Insert #include

A> *esc* – Append > to the end of the line

j – go to the next line (Important)

q – Stop recording macro

Keyboard Macro Example

After recording use the command

@a

to execute it.

To execute 5 times

5@a

Keyboard Macros

- Advantages
  - **Quick**
  - **Simple**
- Disadvantages
  - **Limited**
  - **Temporary**
  - **Impossible to Edit (almost)**

Mapping

Make a mapping out of the macro

```
:map <F11> ^i#include
    <lt><ESC>A><ESC>j
```

(one line)

<F11> -- Key name

Rest of the line is the macro

<lt> -- Less than (there is no <gt>)

Making it Permanent

- To automatically define this mapping when Vim starts put it in the file

  `$HOME/.vimrc`

Modes and Mapping

- **:cmap** – Command line mode
- **:imap** – Insert mode only
- **:lmap** – Inputing a language dependent argument for command mode or insert mode.
- **:map** – Normal, visual, select, operator pending.
- There's more

Modes and Mapping

- **:map!** – Command line and insert
- **:nmap** – Normal
- **:omap** – Operator pending
- **:smap** – Select
- **:vmap** – Visual and select
- **:xmap** – Visual

Mapping and modes

How the different mappings work

- Normal mode

```
:map <F11> ^i#include
 <lt><ESC>A><ESC>j
```

- Insert mode

```
:imap <F11> <ESC>^i#include
 <lt><ESC>A>
```

Adding it to the menu

Adding it to the menu of *gvim*.

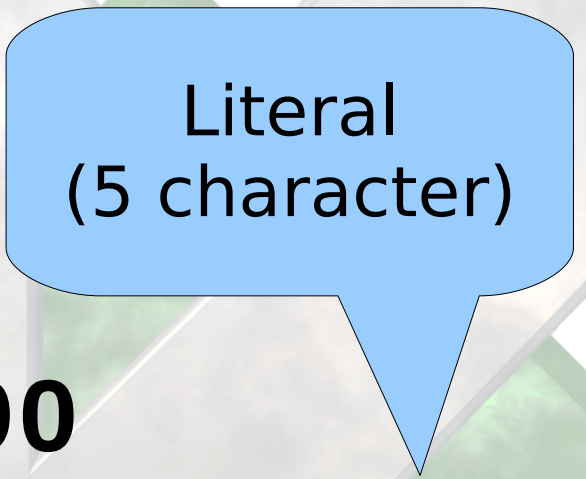
```
:menu 40.290
 &Tools.&Include<Tab>F11
<F11>

(one line)
```

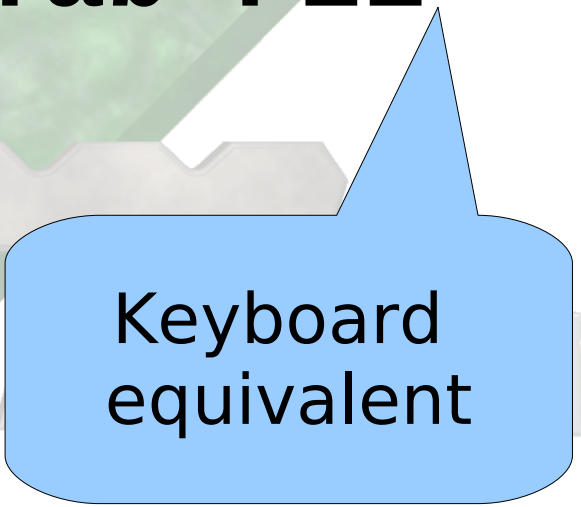
Adding it to the menu


```
:menu 40.290
 &Tools.&Include<Tab>F11
<F11>
(one line)
```

Adding it to the menu




```
:menu 40.290
 &Tools.&Include<Tab>F11
<F11>
(one line)
```



Finding Out What's in a Menu

- List all menu items

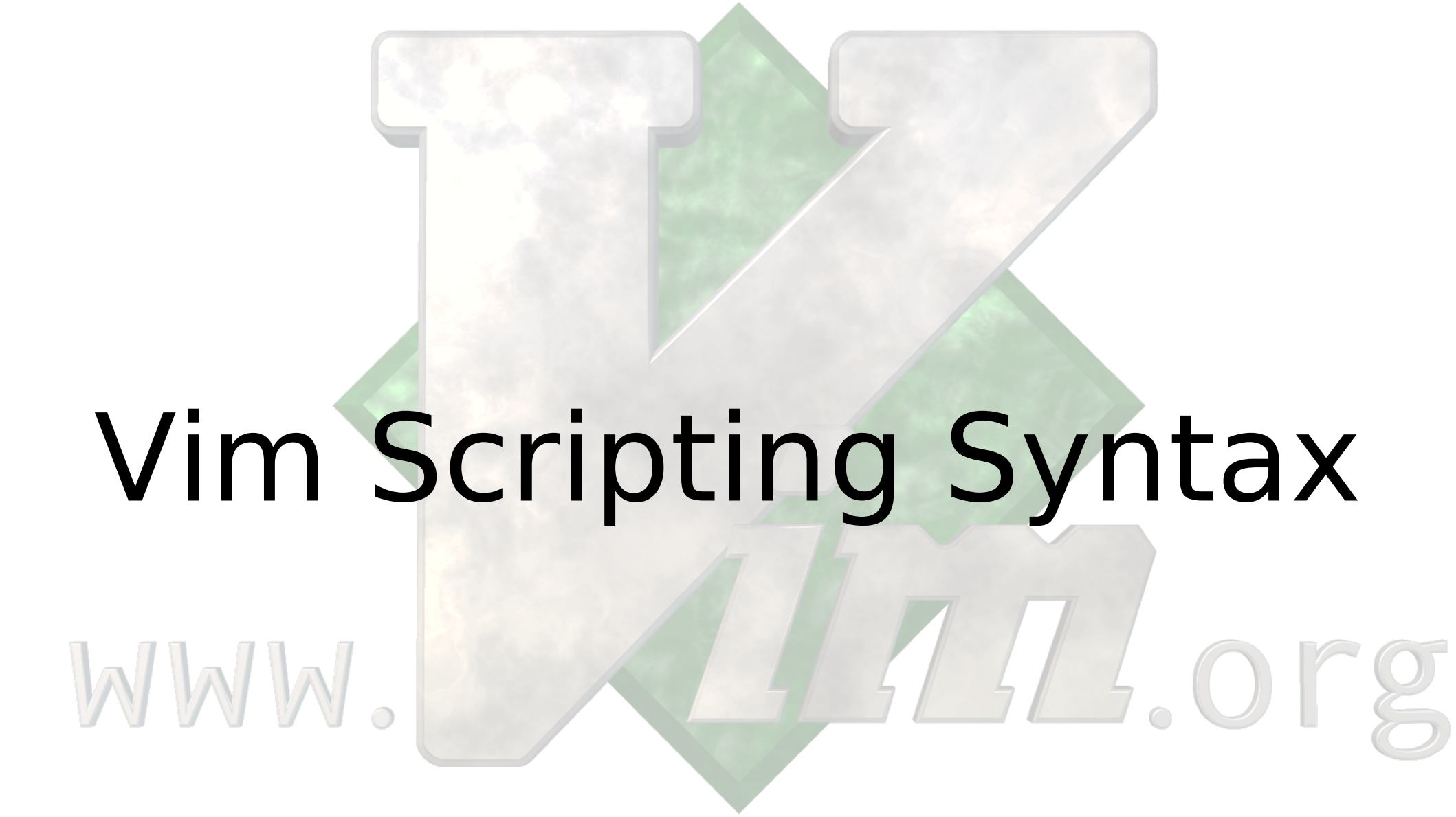
```
:menu
```

- List the Tools menu

```
:menu Tools
```

- List only our entry

```
:menu Tools.Include
```



Vim Scripting Syntax

Cheat Sheet

- We're now going through the file:

  `i-call.vim`

Variable Types

- Variable names follow the usual rules (case sensitive)
- Assignment

```
:let foo = “hello”
:echo foo
```

Prefixes denote name space

<none> -- In a function, local. Outside a function, global.

b: -- Buffer specific

w: -- Window Specific

t: -- Tab Specific

g: -- Global

Prefixes denote name space

l: -- Local to a function

s: -- Local to a script

a: -- Function argument

v: -- System defined variable

Other types of “variables”

&option – The value of an option. The local version is checked first, the the global.

&l:option – Local version of the option.

&g:option – Global option

@register – Register

$ENV – Environment variable

Variable Variable Names

- If {} are used in a variable name, the value of the variable inside the {} becomes part of the name:

```
:let sam_name = “Sam”
:let joe_name = “Joe”
:let who = “joe”
:echo {who}_name
```

Expression Syntax

- Mostly the usual operators (+, -, *, etc.)
- Regular Expression comparison

```
str =~ “re”
```

String Comparison

- String compare, case insensitive

```
str ==? “value”
```

- String compare case sensitive

```
str ==# “value”
```

- String compare, maybe ignore case (depending on 'ignorecase' option)

```
str == “value”
```

Sub-string expressions

- Single character

```
echo str[5]
```

- Substring

```
echo str[3:5]
```

- Next to last character

```
echo str[-2:-2]
```

- Character 5 on

```
echo str[5:]
```

“Include” Macro As a Function

```
:function! Include()
:    " Get the current line
:    let l:line = getline(".")

:    " Put the #include in the right place
:    let l:line = "#include <".l:line.">"

:    " Replace the line
:    call setline(".", l:line)
:endfunction
```

Defining the Function

- Functions names must begin with a capitol.
- The force (!) operator allows us to redefine an existing function.

```
:function! Include()
....
:endfunction
```

Get the line

- Comments begin with “. Makes it hard to comment a string assignment.
- “.” is the current line. Assign it to a local variable (l:line)

```
: " Get the current line
: let l:line = getline(".")
```

Add on the #include stuff

- The dot (.) operator concatenates strings.

```
:" Put the #include in
:” the right place

:let l:line =
\    "#include <".l:line.">"
```

Replace the line with the new one

- Replace the line with the new one.
- Again “.” is the current line number.

```
: " Replace the line
: call setline(".", l:line)
```

Calling the Function

- To call the function, type:

```
:call Include()
```

- But that's too difficult, so let's map it:

```
:map <F11> :call Include()<CR>
```

Initializing Vim's GUI

- We will be putting our macro in the top level menu. This must be done before the menu is built.
- To run a script before the GUI is done:

gvim -U *macro-file file*

Initializing Vim's GUI

- All top level menu commands are ignored after the GUI is built.
- If you want it to load automatically put it in:

  $HOME/.gvimrc

- WARNING: Do not put it in .vimrc, it won't work

Diversion: Initialization Problems

- Vim starts in *Vi compatibility* mode. (Yuck)
- In *Vi* mode <F11> is 5 characters, not a function key.

Initialization Solutions

- Save the compatibility options, then set them to the *Vim* defaults

```
" Save options
:let s:cpo_save = &cpo
:set cpo&vim
```

- … at the end restore them

```
:let &cpo = s:cpo_save
:unlet s:cpo_save
```

Problem: -U skips .gvimrc

- The file you specify with -U skips the .gvimrc
- Solution #1. Two -U
  - **`gvim -U m.vim -U ~/.gvimrc`**

Problem: -U skips .gvimrc

- Solution #2. Source it at the end of your file:

```
:let &cpo = s:cpo_save
:unlet s:cpo_save
:source ~/.gvimrc
```

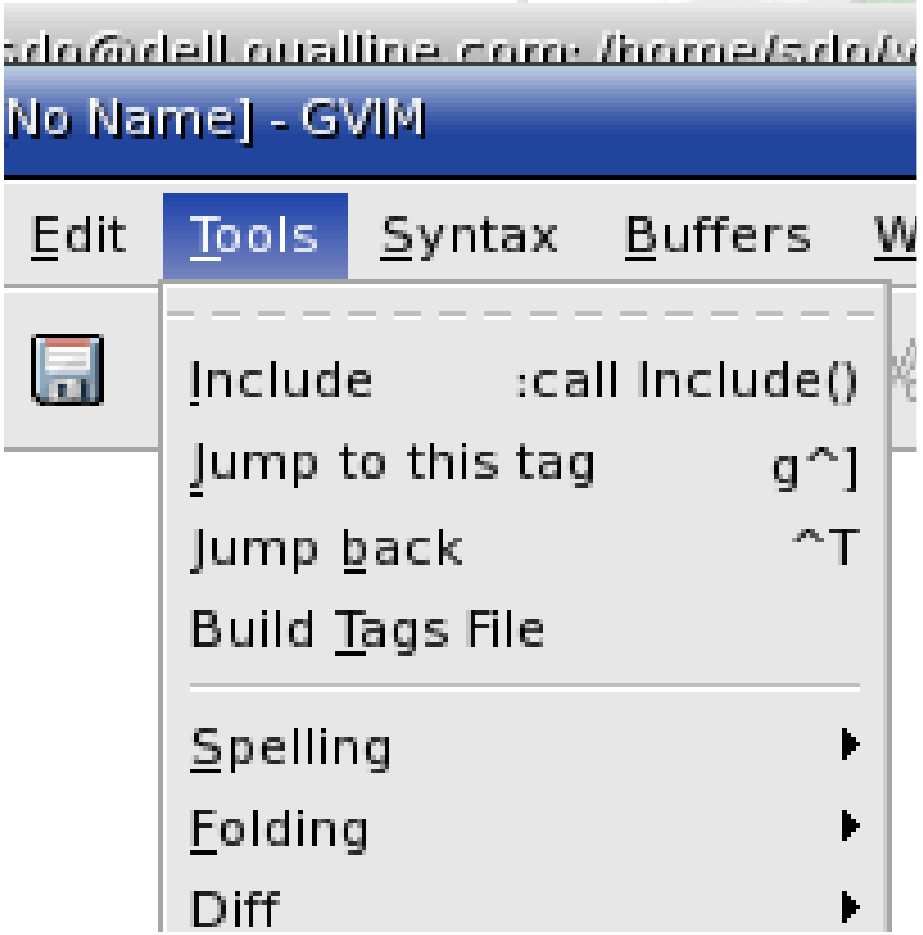
Putting the command in the menu

```
:menu 40.290
\ &Tools.&Include<Tab>
\ call\ Include()
\ :call Include()
```

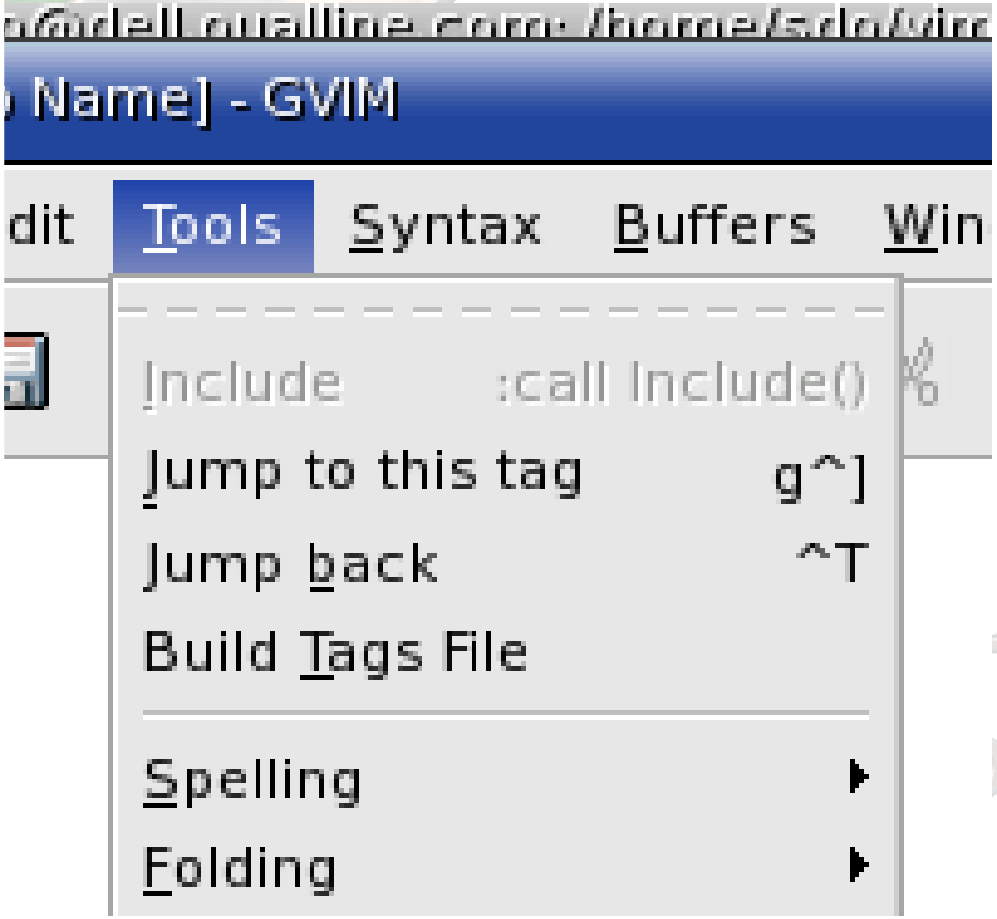
- But there is a problem. The commands disappear in some modes

The Mode Problem

- Normal Mode



- Insert Mode



Let's look at what :menu did

```
:menu Tools.Include
--- Menus ---
290 &Include^I:call Include()
    n    :call Include()<CR>
    v    :call Include()<CR>
    s    :call Include()<CR>
    o    :call Include()<CR>
```

Define a menu item for all modes

```
:amenu 40.295
\ &Tools.&Include(a)<Tab>
\ call\ Include()
\ :call Include()
```

Our menu looks different

```
:amenu Tools.Include(a)
--- Menus ---
295 &Include(a)^I:call Include()
    n    :call Include()<CR>
    v    <C-C>:call Include()<CR><C-\><C-G>
    s    <C-C>:call Include()<CR><C-\><C-G>
    o    <C-C>:call Include()<CR><C-\><C-G>
    i    <C-O>:call Include()<CR>
    c    <C-C>:call Include()<CR><C-\><C-G>
Press ENTER or type command to continue
```

What's happening

v <C-C>:call Include()<CR><C-\><C-G>

V – Visual mode

<C-C> -- Exit visual mode

:call Include() -- The command

<C-\><C-G> -- Back go previous mode

What's happening

`i <C-O>:call Include()<CR>`

i – Insert mode

<C-O> -- Execute a single normal mode code, then go into insert mode.

:call Include() -- The command

Adding it to the popup menu

- Add the call to the menu which pops up when you press the right menu button.

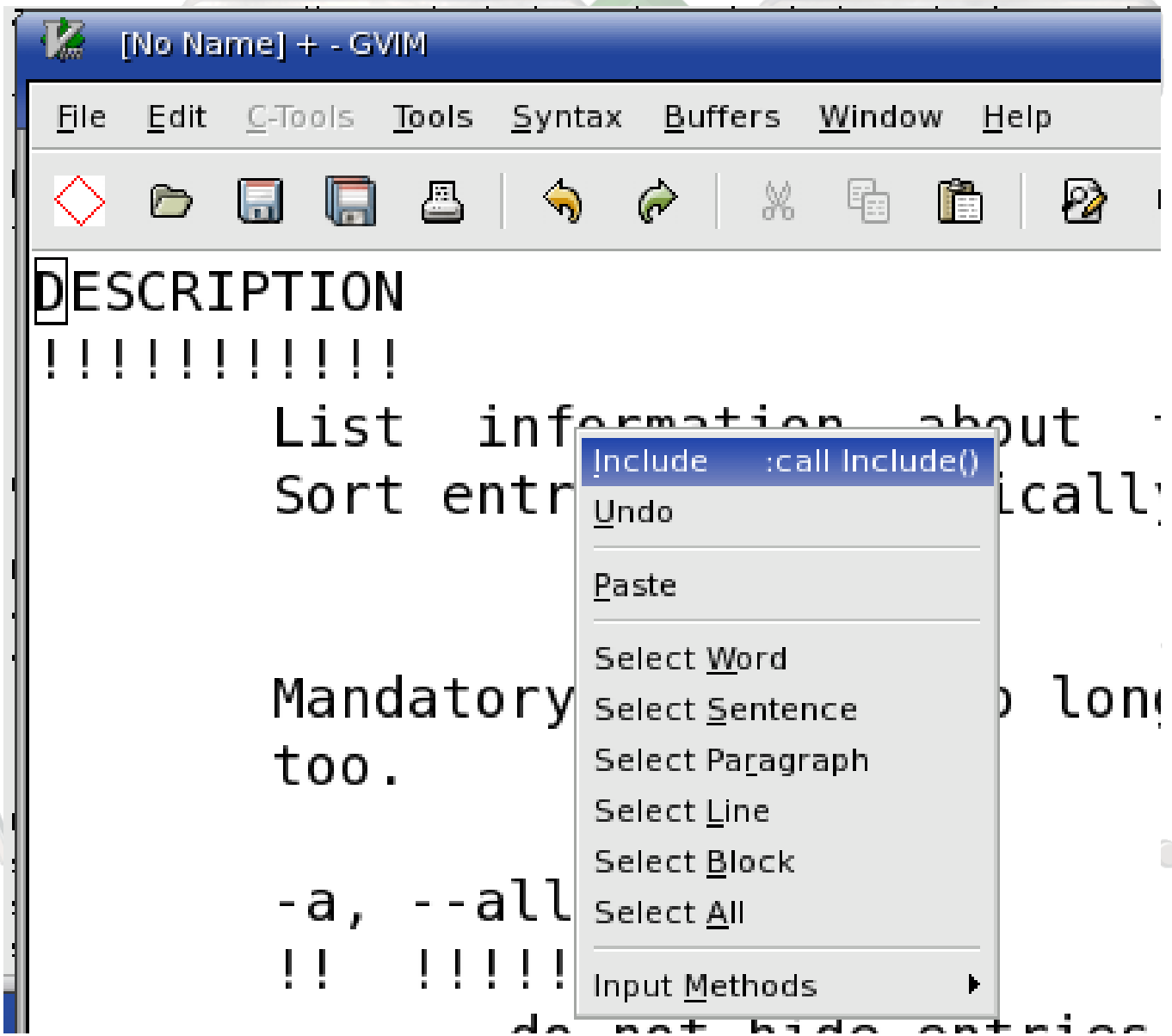
```
:amenu 1.5
  PopUp.&Include:call\
  Include() :call Include()
```

(one line)

- WARNING: You must have enable the popup

```
:set mousemodel=popup
```

The new popup menu



Putting the command in the toolbar

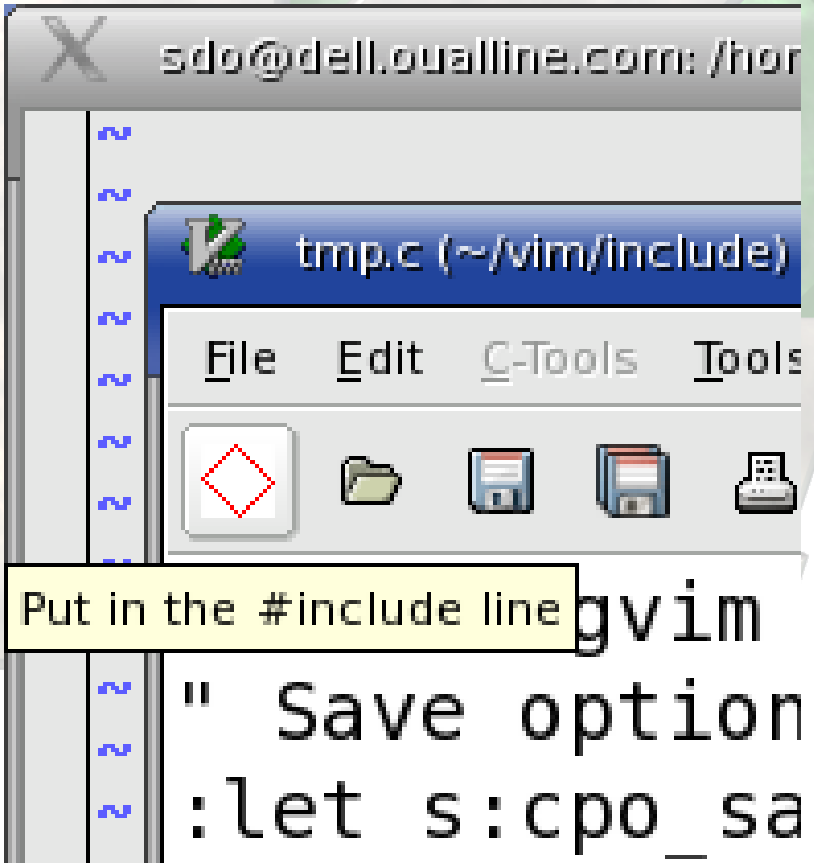
- The menu “Toolbar” is the top level tool bar.
- You can include an icon specification in the menu command as well as the normal stuff.

```
:amenu icon=/home/sdo/vim
 /include/include.xpm 1.1
ToolBar.Include
:call Include()
```

(one line)

Adding a tool tip to the toolbar icon

```
:tmenu ToolBar.Include
\ Put in the #include line
```



Creating a top level menu

- Creates a top level menu C-Tools with a single item

```
:amenu 30
\   &C-Tools.Include<Tab>F11
\   :call Include()
```

Enabling and disabling the menu

- Enable

```
:menu enable &C-Tools
```

- Disable

```
:menu disable &C-Tools
```

The function to enable or disable the C-Tools menu

Depending on file type (&ft) enable the menu

```
:function CMenuCheck()
:    if ((&ft == "c") || (&ft == "cpp"))
:        :menu enable &C-Tools
:    else
:        :menu disable &C-Tools
:    endif
:endfunction
```

Automatically Calling the function

- Automatically call the function when a buffer is entered.

```
:autocmd Bufenter *
\     :call CmenuCheck()
```

- Call it when the file type changes

```
:autocmd FileType *
\     :call CMenuCheck()
```

Cheetsheet time

- We've now moved on to:

  `i-cmd.vim`

Creating a new command

- Defining a user command to do the includes

```
:command! -nargs=0
-range Include
:call IncludeRange(
<line1>, <line2>)
```

(one line)

Defining a command

:command! -- Define a user command

-nargs=0 – Number of arguments

-range – Can take a line range as input

Include – Name of the command (User commands must start with upper case letter)

:call.... -- Command to execute

<line1>, <line2> -- Start / Ending lines for the command.

Definition of IncludeRange with debugging code

```
:function! IncludeRange(first_line, last_line)
:    let l:cur_line = a:first_line
:
:    while (l:cur_line < a:last_line)
:        call setpos('.', [0, l:cur_line, 0, 0])
:    call Include()
" Debug stuff
:echo l:cur_line
:redraw
:sleep 5
:        let l:cur_line = l:cur_line + 1
:    endwhile
:endfunction
```

Starting off

- Function takes two arguments, a first line and a last line.

```
:function! IncludeRange(first_line, last_line)
```

- Define a variable to loop through the lines

```
:    let l:cur_line = a:first_line
```

Move the Cursor to the given line

- Loop through each line

```
:    while (l:cur_line < a:last_line)
```

- Move the cursor ('.') to the given [buffer, line, character, offset]

```
:        call setpos('.', [0, l:cur_line, 0, 0])
```

- Call the Include() function

```
:    call Include()
```

Debug Stuff

- Print the current line
- Redraw the screen (show partial progress)
- Sleep for 5 seconds (to make sure we can see what happened)

```
" Debug stuff
:echo l:cur_line
:redraw
:sleep 5
```

Finishing Up

- Finishing up

```
:          let l:cur_line = l:cur_line + 1
:      endwhile
:endfunction
```

Defining a better command

```
:command! -nargs=0
-range Include2
<line1>, <line2>:call
 Include()
```

(one line)

- If a **<range>** is specified for **:call,** then the function is called once for each line.

Review: What we can do with functions

1. Call them directly (**:call**)
2. Map them to a key (**:map**)
3. Put them in the menu (**:amenu**)
4. Put them in the toolbar (**:amenu toolbar**)
5. Put them in the popup menu (**:amenu popup**)
6. Create a user command to call them (**:command**)

Improving the Include function

- Checks local directories for the file
- If local puts in #include “file.h”
- Checks system directories
- If found puts in #include <file.h>

Cheat Cheat

- Moving on we reach the cheat sheet

  `i-fancy.vim`

Improved Include function

First some definitions

```
" System include dirs
:let g:SystemIncludes = [
\     "/usr/include",
.....
\]
" Local includes follow
:let g:LocalIncludes = [
...
```

Define the function

- Some starting code to get the line

```
:function! Include()
:    " Get the current line
:    let l:line = getline(".")
```

Loop through the dirs

```
:     for l:cur_dir in g:LocalIncludes
:         if (filereadable(l:cur_dir."/".l:line))
:             let l:line =
\                     "#include \"".l:line."\""
:             call setline(".", l:line)
:             return
:         endif
:     endfor
```

- Do the same thing for the system dirs

Debugging the function

- To start the debugger

  :debug call Include()

- Debugging commands

  :echo – Display expression

  :step – Single Step

  :next – Skip over function

Debugging the function

- Setting a breakpoint

```
:breakadd func <line>
 <function>
```

Debugging Commands

- Running a command with the debug:

```
:debug call Include()
```

- Turning on the verbose chatter:

```
:16verbose call Include()
```

- Setting the verbose level:

```
:set verbose=16
```

Saving the output

- To log the output

```
:redir! >log.txt
```

- To stop logging

```
:redir END
```

The Configuration Problem

- We must configure the thing by setting two global variables.

- There is a Vim option called path. Why can't we use that?

Cheat Sheet

- Almost done with include. Take a look at:

  `i-path.vim`

Revised Function

- Turn the path option (&path) into a list of directories.

```
:let l:dir_list =
\       split(&path, ",")
```

Revised Function

- Loop through each entry looking for the file

```
:for l:cur_dir in l:dir_list
:if (filereadable(
    l:cur_dir."/".l:line))
```

Revised function

- System directory?

```
if (match(l:cur_dir,
\  "/usr/include") == 0)

:    let l:line =
\ "#include <".l:line.">"
```

Demonstration of Fancy Include Function

GUI Version of the Include

- This version asks you which type of include (system, local) you want and does the work accordingly.

Cheat Sheet

- The cheat sheet for this is

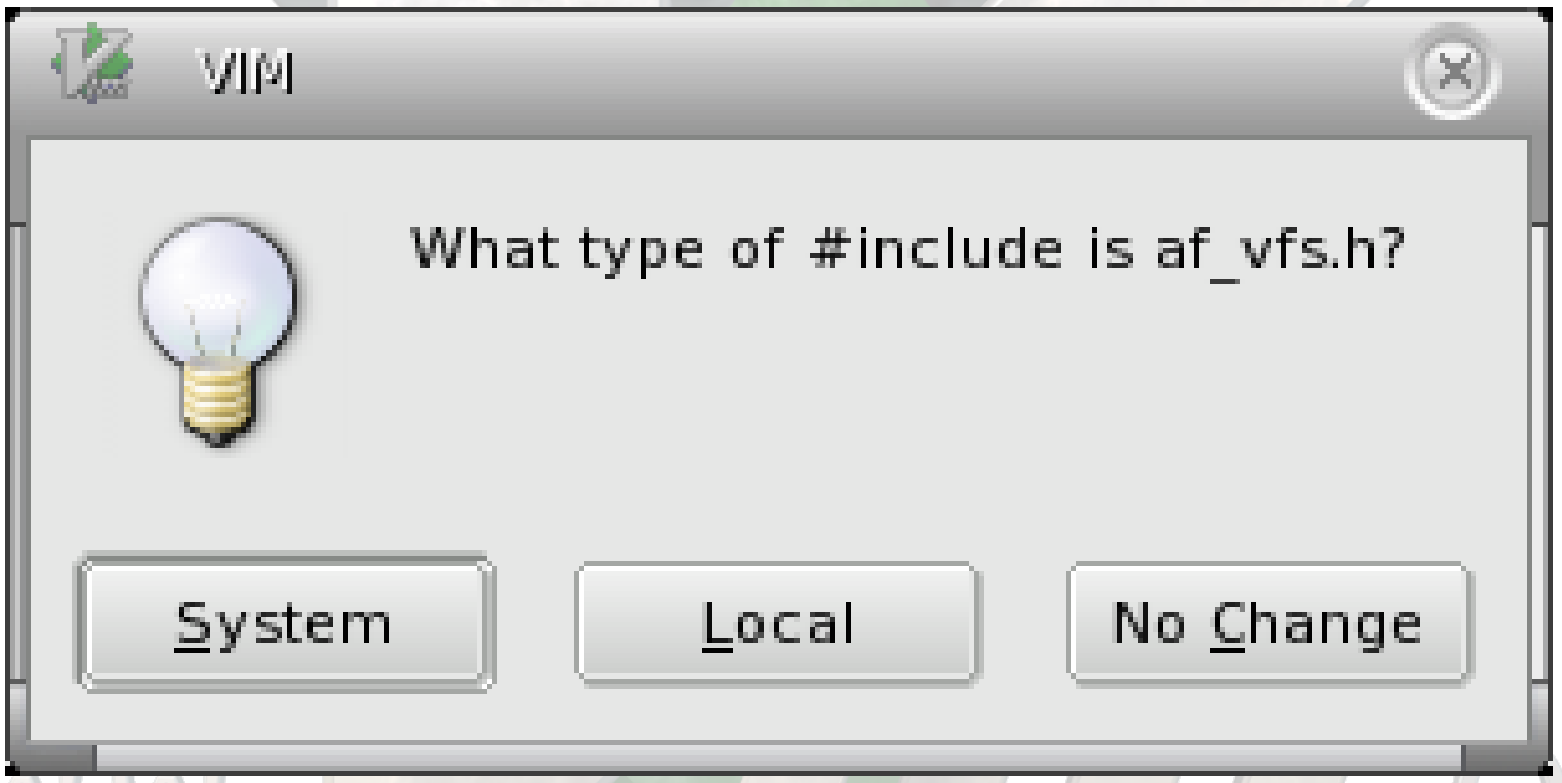
  `i-gui.vim`

Displaying a Dialog Box

- Let's display a dialog box with three choices:

```
:    let l:choice =
confirm("What type of #include
 is ".l:line."?",
"&System\n&Local\nNo \&Change")
```

The Dialog Box



confirm() return values

1. System
2. Local
3. No Change
0. Dialog was closed manually

Dealing with the result

```
:    if (l:choice == 1)
:        let l:line = "#include <".l:line.">"
:        call setline(".", l:line)
:        return
:    elseif (l:choice == 2)
:        let l:line = "#include \"".l:line."\""
:        call setline(".", l:line)
:        return
:    elseif (l:choice == 3)
:        return
:    elseif (l:choice == 0)
:        throw "WARNING: You closed the dialog!"
:    else
:        throw "ERROR: There is no choice
  ".l:choice."  Huh?"
:    endif
```

Demonstration

GUI Based #include generator

Congradulations

#include is now exhausted

Java Editing Function

- Define a function to make Java editing easier
- Automatically adds the “getter” for a java program.

Cheat Sheet

- Our first java cheat sheets:

  `g1.vim`

  `bean.java`

Java – Adding the getter

- What we have

```
class bean {
    private int value;
};
```

What we want

```
class bean {
    private int value;

    public int getValue() {
        return (value);
    }
};
```

Algorithm

1. Parse the line under the cursor.
   Determine the variable's name and type.
2. Search backward for “class”
3. Search forward for “{“
4. Finding matching “}”
5. Insert the getter code.

1. Parsing the line

```
:function! Getter()
:    " Get the line defining the variable
:    let l:var_line = getline('.')

:    let l:prot = substitute(l:var_line,
  '\v^\W*(\w+)\W+.*$', '\1', '')

:    let l:type = substitute(l:var_line,
  '\v^\W*\w+\W+(\w+)\W+.*$', '\1', '')

:    let l:var  = substitute(l:var_line,
  '\v^\W*\w+\W+\w+\W+(\w+).*', '\1', '')
```

Notes on step 1.

- Can you make the regular expressions a little more complex?

A: Absolutely You should see my talk on regular expressions.

- What do they all mean?

A: Use the **:match** command in *Vim* to find out. It highlights text matched by a regular expression.

Regular Expression Exploded

\v – Make all the following characters special except digits and letters

^ -- Start of line

\W* -- Whitespace (\W) zero or more times (*)

(\w+) -- Place matching item in \1. Word characters (\w), one or more times (+)

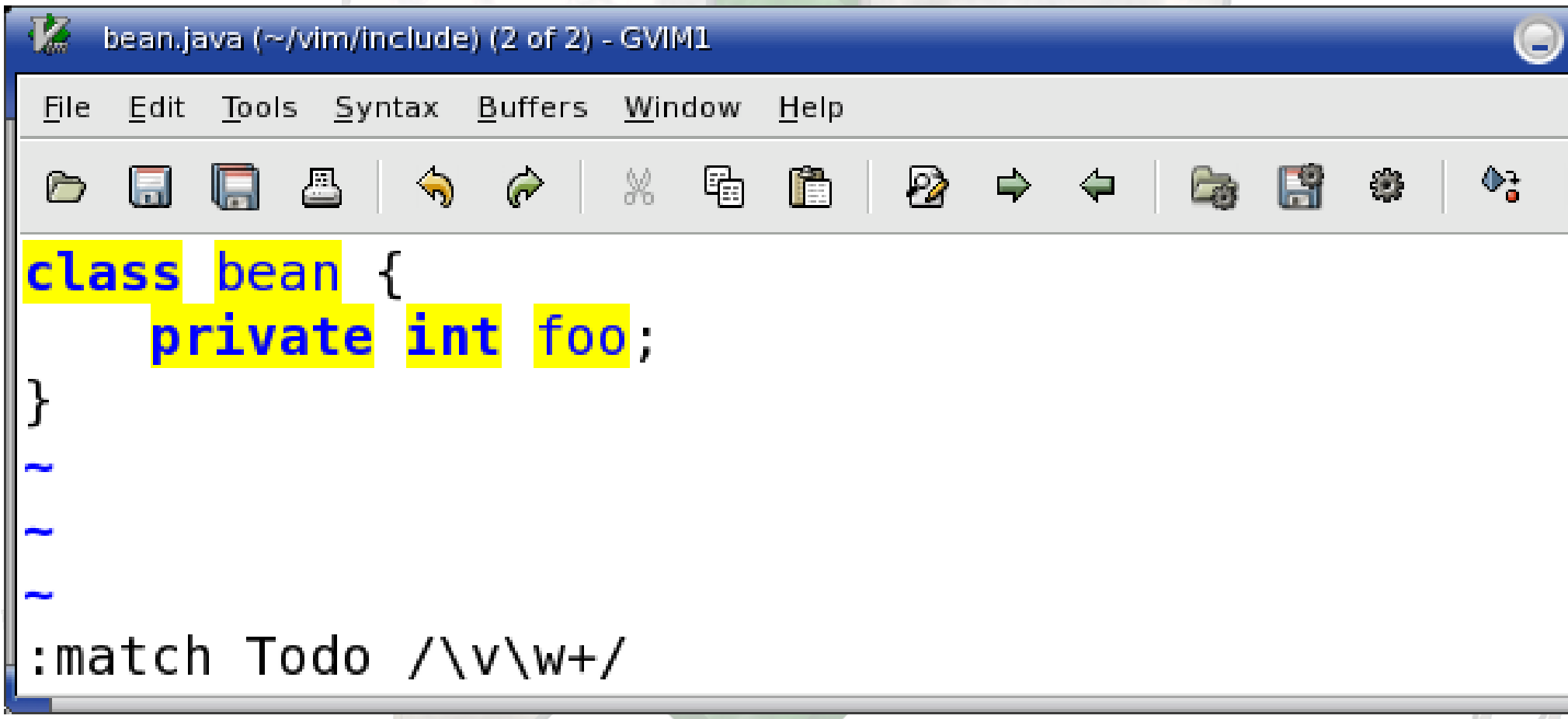
Regular Expression Exploded

\W+ -- Whitespace (\W) one or more times (+)

.* -- Any character (.) zero or more times (*)

$ -- End of line.

A short demonstration of :match



Sanity Checking

```
:if ((l:prot != 'public')
\ && (l:prot != 'private')
\ && (l:prot !=
\                 'protected'))
:    throw "ERROR: Unable
 to parse variable line"
:endif
```

2. Reverse search for “class”

```
:if (search(
\       'class', 'b') == 0)
:   throw "ERROR: ...."
:endif
```

3. Forward search for “{“

```
:if (search(
            '{', '') == 0)
:    throw "ERROR: ..."
:endif
```

4. Finding the matching “}”

%

- Simple wasn't it.

5. Compute the function name

- Substitute in l:var
- Everything (.*)
- Make next (first) character upper case (\u) and then everything else the same (&)

```
:    let l:fun_name =
substitute(l:var,
                '.*',
               '\u&', '')
```

Computing the text to insert

```
:let l:getter = ["     /**",
\ "       * Get the current value of ".l
\ "       *",
\ "       * @returns ".l:var,
\ "       */",
\ "      public ".l:type." get".l:fun_na
\ "          return(".l:var.");",
\ "      }"
\ ]
```

Insert the text

```
: let l:where = line('.')
: let l:where = l:where - 1
: call append(
\             l:where, l:getter)
```

Demonstration

Live Demo of Getter Version 1

For those who came in late

- I am Steve Oualline
- Slides and cheat sheets at
  - **www.oualline.com/vim.talk**

But what about mean programmers?

```
class mean {
      /* comment with
       * “class” in it
       * and “}” as well
       */
     private int value;
```

Cheat Sheet Time

- We now go to the nasty bean and the getter for it.

mean.java

g2.vim

Fixing the problem

- We don't have to get clever because *Vim* already is.
- We just need to look for “class” which is syntax colored as a Java keyword.
- To find the name syntax item the cursor is sitting on:

```
:echo synIDattr( synID(
\  line('.'), col('.'), 1),
\  'name')
```

Function synID

synID(*line, col, trans*)

- line – Line number of the item
- col – Column number of the item
- trans – If set transparent items are reduced to the base syntax

Function synIDattr

synIDattr(*id, what*)

- id – The syntax id number
- what – What to get ('name' is the name of the item.)

Revised “class” search

```
:while (1)
:  if (search('class', 'b') == 0)
:    throw 'ERROR: Could .....
:  endif
:  if (synIDattr(
\      synID(line('.'), col('.'), 1),
\          'name') == 'javaClassDecl')
:    break
:  endif
:endwhile
```

getter Limitations

- Does not handle complex types such as:

```
java.util.Map foo;
```

- Solution: Better regular expressions

- Does not indent the function properly
- Solution: Use Vim's **:indent** command.

Vim and Perl Programming

First Lesson of Programming

- Remember Vim has a filter command (!) that let's you filter text through an external program.
- Use that and you don't have to mess around with Vim programming

Introducing the Players

- **tab.pm –** Module containing one function: make_tab.
    - Turns a series of lines into column aligned lines.
- **tab.pl –** Program that passes all it's input through make_tab.
- **perl.vim** – Interface module between Vim and tab.pm.

Getting Started

- The program *tab.pl* is a stand alone filter that can be run from the command line.

- We can use it (without special work) from withing Vim by using the filter (!) command.

First step: Vim and Perl

- Perl is not compiled in by default
- Always make sure you have the module before defining a script

```
:if ! has('perl')
:    throw "ERROR: This version of Vim has no Perl feature"
:endif
```

Perl Related Commands

- Putting Perl Code in Vim

```
:perl <<EOF
... perl code ...
EOF
```

- Filter a set of lines through perl

```
:<range>perldo <command>
```

Perl / Vim Interface

- Output a message

```
VIM::Msg("Hello World");
```

- Vim also supplies you with a set of Window and Buffer objects with which to play width:

```
my $window = $main::curwin;
my ($row, $column) =
        $window->Cursor();
```

Getting help

- For information about the Vim/Perl interface:

```
:help :perl
```

```
:perl <<EOF
# Real work done here
require 'tab.pm';

sub tab_lines($$) {
    my $start = shift;
    my $end = shift;
    my $cur_buf = $main::curbuf;

    my @lines =
        $cur_buf->Get($start..$end);
    @lines = make_tab(@lines);
    $cur_buf->Set($start, @lines);
}
EOF
```

Initial work

- Bring in the other module

```
:perl <<EOF
# Real work done here
require 'tab.pm';
```

- Define a function with two arguments

```
sub tab_lines($$) {
    my $start = shift;
    my $end = shift;
```

Process the lines

- Get the lines, push them through the function, put them back in the buffer

```
my $cur_buf = $main::curbuf;
my @lines =
        $cur_buf->Get($start..$end);
 @lines = make_tab(@lines);
 $cur_buf->Set($start, @lines);
```

Now link the perl function to a Vim command

```
:command! -nargs=0 -range Table
\   :perl tab_lines(
\                   <line1>, <line2>)
```

Perl Demonstration

More Perl / Vim Interface

- Global variables

  `$main::curwin`

- The current window object.

  `$main::curbuf`

- The current buffer object.

Message Functions

- Simple Message

```
VIM::Msg("Text")
```

Message with highlighting

```
VIM::Msg(
\    "remark", "Comment")
```

Option Related Functions

- Set option

```
VIM::SetOption("opt")
```

- Getting an option

```
my $opt =
  VIM::Eval(“&opt”);
```

Buffer and Window Information

- Get a list of buffer

```
@buflist = VIM::Buffers()
```

- Get buffers for a specific file

```
@buf =
        (VIM::Buffers('file'))
```

- Get Window List

```
@winlist = VIM::Windows()
```

Window Operations

- Set height

```
$window->SetHeight(10)
```

- Get cursor location (row, column)

```
($row, $col)=
        $window->Cursor()
```

- Set cursor location

```
$window->Cursor($row, $col)
```

- Get the buffer for the window

```
$mybuf = $curwin->Buffer()
```

Buffer Information

- Get the buffer's name

```
$name =$buffer->Name()
```

- Get the buffer's number

```
$number = $buffer->Number()
```

- Get the number of lines in the buffer

```
$lines = $buffer->Count()
```

Buffer Data

- Get a line or array of lines

```
$line=$buffer->Get($number)
@lines = $buffer->Get(
         @number_array)
```

- Setting a line or set of lines

```
$buffer->Set(
      $number, $line)
$buffer->Set($start_number,
    @line_array)
```

Buffer Data

- Deleting lines

```
$buffer->Delete($line)
$buffer->Delete(
        $start, $end)
```

- Adding lines

```
$buffer->Append(
        $number, $line)
$buffer->Append(
        $number, @line_array)
```

Advanced Topics

Dictionaries

- Defining a dictionary variable

```
:let g:dict = {
    “key” : “value”,
    “key2” : “value2” }
```

- Getting a value

```
echo dict[“key”]
```

- Setting a value

```
let dict[“key3”] = “value3”
```

Exceptions

```
:try
:    " do something
:catch /ERROR/
:    " fix something
:finally
:    " Finish up
:endtry
```

Plugins

- Plugins are automatically loaded with Vim starts
- Local Plugins

  ~/.vim/plugin/*file*.vim

- Global Plugins

  $VIMRUNTMIE/plugin/*file*.vim

Special Plugins

- Syntax coloring files

  $VIMRUNTIME/syntax/*lang*.vim

- Indentation functions

  $VIMRUNTIME/indent/*lang*.vim

Autoloading Functions

Please don't autoload
Unless you really
really have to

Autloading

1. Define your functions

   `:function! CallMe() ....`

2. Put them in the file:

   `~/.vim/autoload/file.vim`

3. Call the function using the magic call:

   `:call file#CallMe()`

Improvements Yet to be made

- All scripts can be improved. Here are some things we didn't do in the scripts for this class.

1. Use **findfile()** to locate the #include instead of going through the directory list one at a time.
2. Use expand('<cword>') to get the word under the cursor in our getter function.

Getting Scripts

- The Vim site (http://www.vim.org) contains a link to a script library.
- Lots of scripts of varying quality are available

Irony

- This class presented many many different ways of automatically generating #include lines.
- What way does the author use?

```
:ab #i #include
```

- When #i is typed in Vim, #include will be inserted.

Finally: Remember

Vim is Charity-Ware

Please Donate

ICCF Holland - helping children in Uganda

Vital Information

1) Please fill out your evaulations.

   Vim Class / Steve Oualline

   www.orielly.com/go/os07tuteval

2) Course Materials Can Be Downloaded from:

   http://www.oualline.com/vim.talk

3) Donate. For Information, start *Vim* then enter the command:

   :help uganda