Training course

JavaScript for Oasys PRIMER and D3PLOT

24th June 2016

Contents

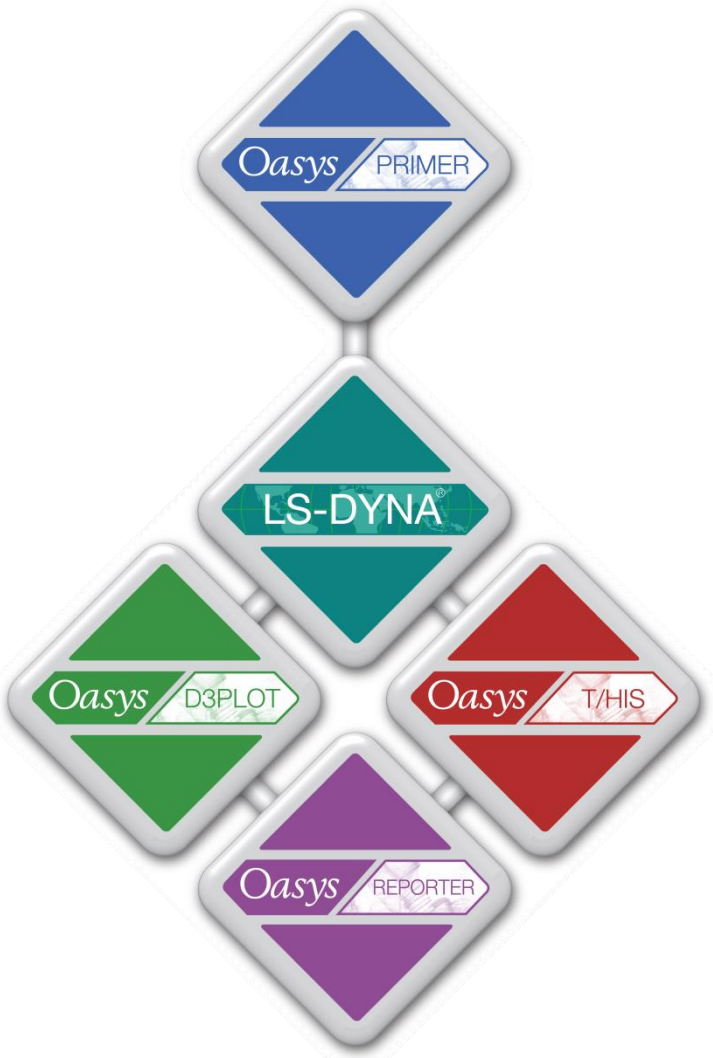
Aims of this course

- Familiarisation with the JavaScript language – it is not expected that participants will already know JavaScript.
- Learning how to write and run JavaScripts in PRIMER and D3PLOT
- For those wishing to study only PRIMER JavaScripts, the section on JavaScripts for Oasys D3PLOT may be omitted.

Which Oasys software products have JavaScript?

JavaScript is now available in all Oasys Software products.

- Oasys PRIMER
- Oasys D3PLOT
- Oasys T/HIS
- Oasys REPORTER



What is JavaScript?

- Fully-featured programming language, widely used for web programming
- JavaScript has “Core” (standard) capabilities described in textbooks
- JavaScript Interpreter can be embedded in other software, e.g. PRIMER and D3PLOT
- The Oasys software development team can extend JavaScript by adding classes and methods for communication with PRIMER, D3PLOT and T/HIS’s data and capabilities.
- The user’s scripts can include both Core and Oasys extensions.
- The compilation step is done inside the interpreter – the script is source code and works on any computer platform.
- The interpreter is included inside PRIMER, D3PLOT and T/HIS – no special software or system setup is required.

Why write JavaScripts?

Advantages of writing a JavaScript to create a new capability:

- Quick turnaround – don't have to wait for a new version of T/HIS, D3PLOT or PRIMER to be released
- Can keep your application confidential
- Under your control – can build it yourself if you wish.

Example applications (PRIMER):

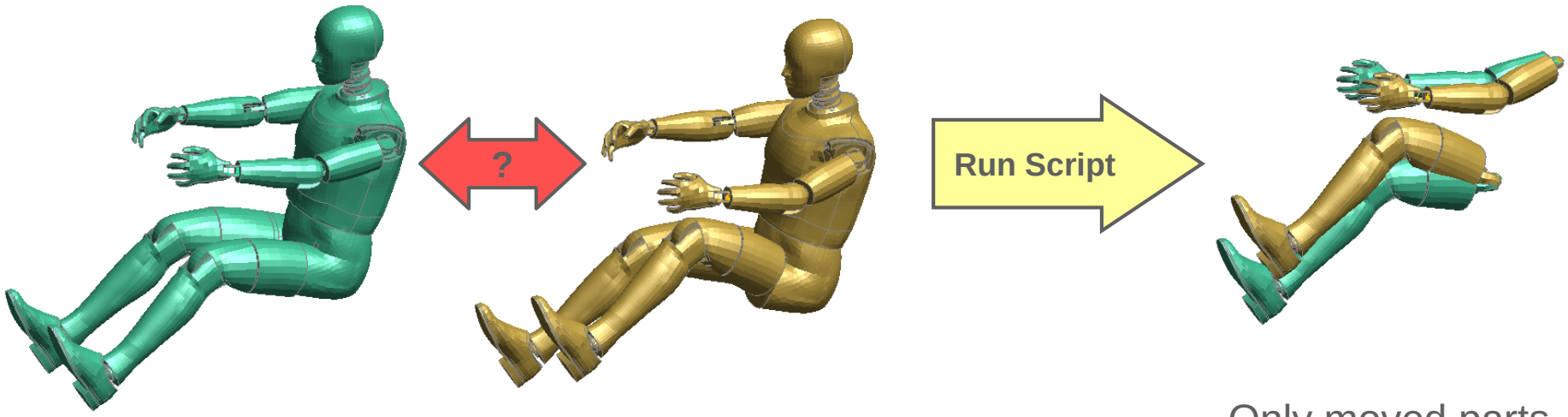
- Creating a simple mesh, or test models with standard loading
- Data checking or correcting
- Geometric morphing functions
- Input or output translators, special-format spotweld or connections files
- Automating routine tasks

Example applications (D3PLOT):

- Generating your own data components for plotting, calculated from any information already contained in the model or from external files

Example application: Finding moved parts

- Available in the from the file tree listing in the Scripts menu, “find moved”
- This script compares two similar models, and unblanks only those elements whose nodes have different coordinates between the two models.



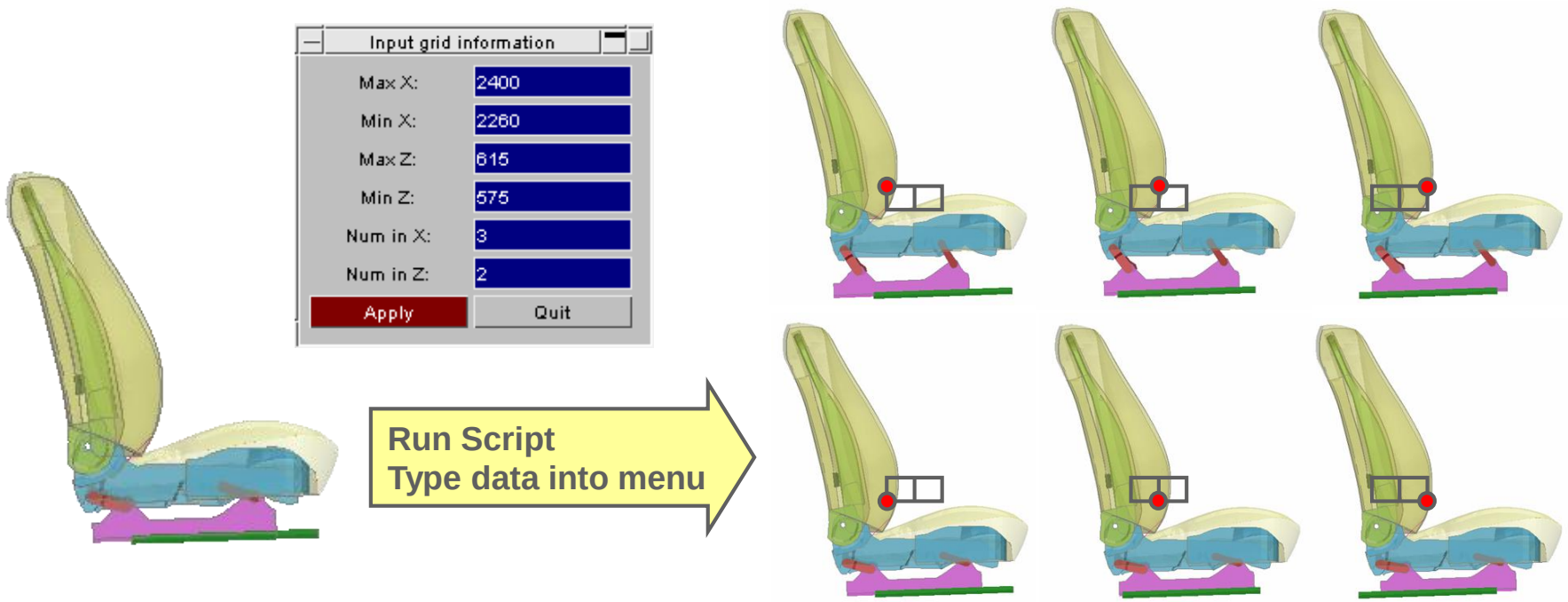
Model 1

Model 2

Only moved parts are displayed

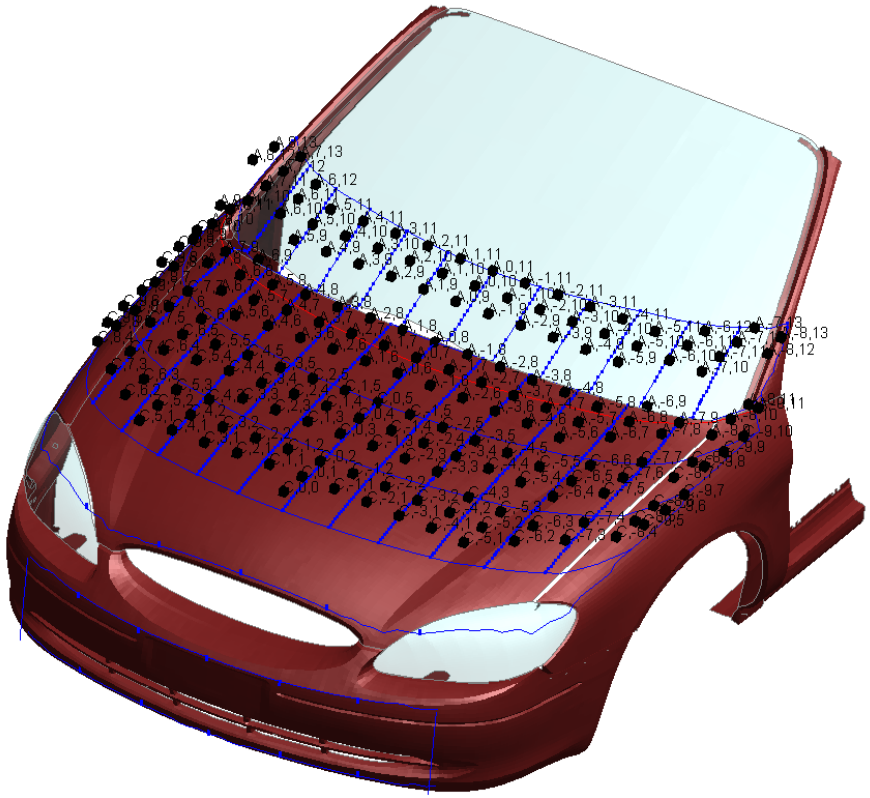
Example application: Multiple seat position

- This script *seat_position.js* is available in the examples directory *$OASYS/primer_library/examples*
- Creates multiple seat models with the H-point in different positions.
- The script includes a menu window (GUI) so the user can type in data to make a rectangular grid of H-point positions.



Example application: Pedestrian Impact Zone Setup

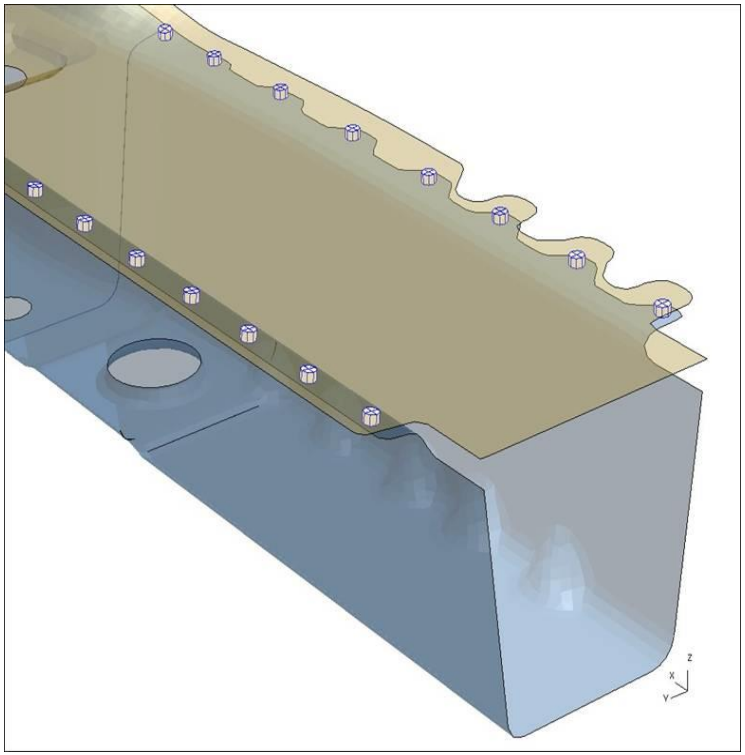
- This script *pedestrian_impact_marking_program.js* is available in the scripts directory *$OASYS/primer_library/scripts*
- This is used to calculate pedestrian impact zone boundaries and impact points.



Example application: Read custom-format spotweld file

Custom-format spotweld file

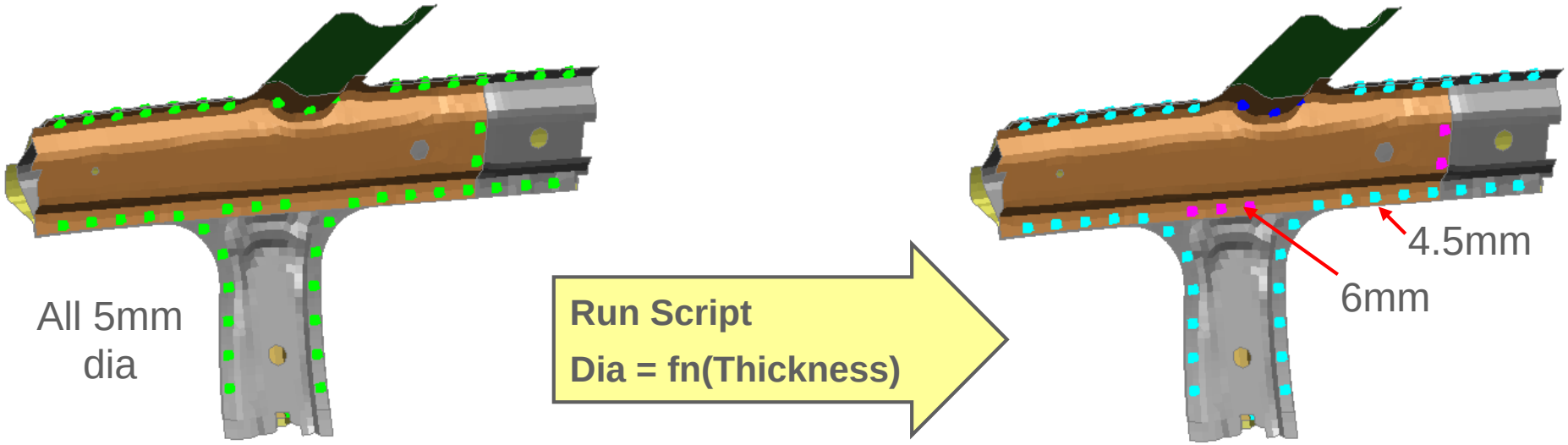
```
<welds>
  <weld>
    <coord>3734.050293 586.282166 2347.783936</coord>
    <pid>82151</pid>
    <pid>8700</pid>
  </weld>
  <weld>
    <coord>3694.061523 586.860229 2347.063721</coord>
    <pid>82151</pid>
    <pid>8700</pid>
  </weld>
  <weld>
    <coord>3654.075928 587.419556 2346.248291</coord>
    <pid>8700</pid>
    <pid>82151</pid>
    <pid>8710</pid>
  </weld>
```



Run Script → **Read in the file** → **Sort the data** → **Create Spotwelds**

Script to apply spotweld properties

- Spotweld diameter, failure properties, etc are a function of sheet thickness and yield stress
- Each customer may have their own different function for calculating these
- A Script could adjust the spotweld size and failure properties using this function

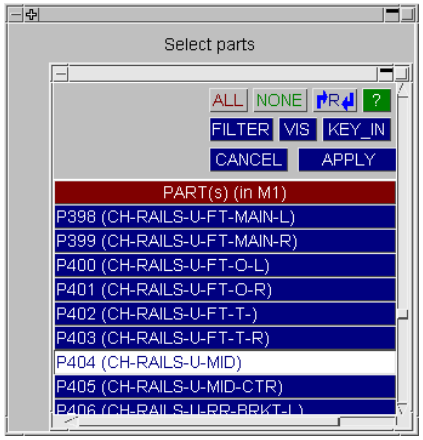


Example application: Interfacing with other programs

This script asks the user to select parts, runs an external program (HYCRASH from JSOL Ltd) to perform 1-step metalforming analysis, and imports the resulting thickness, stress and strain data into the model.

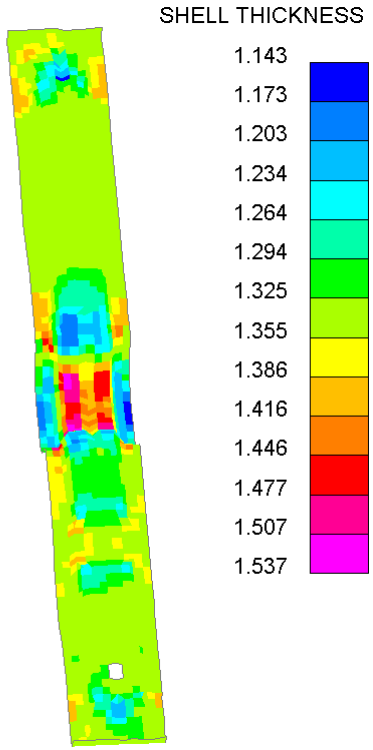
Crash model

Run Script



Select parts in crash model

Script runs HYCRASH



Crash model now has forming data

Automatic model assembly

Microsoft Excel - Book1

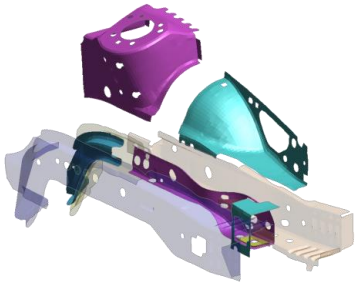
File Edit View Insert Format Tools Data Window OvaE

Reply with Change

J33

	A	B	C	D	E
1	CASE	SPEED	ANGLE	DUMMY	OUTPUT
2	FMVSS208				
3	FMVSS208				
4	ODB				
5	Pole				
6	Euro Side				
7	JNCAP				
8	SINCAP				
9	15mph				
10	Canadian				
11	Rear				
12	FMVSS214				

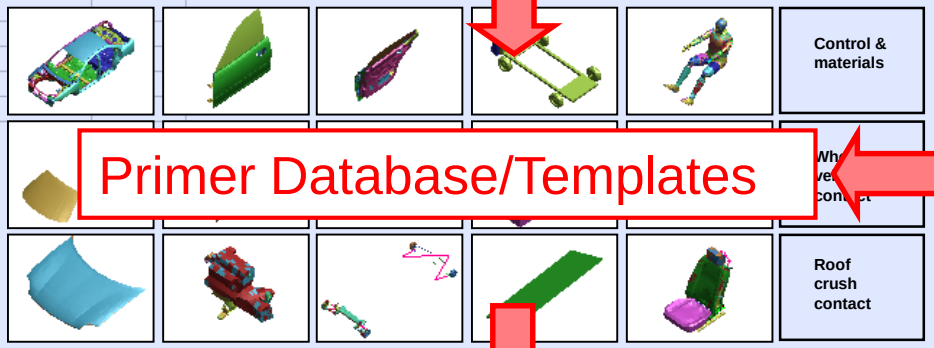
Customer's unique input



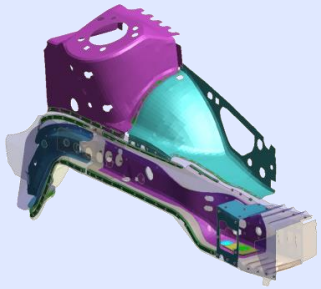
Script

Read

Connect



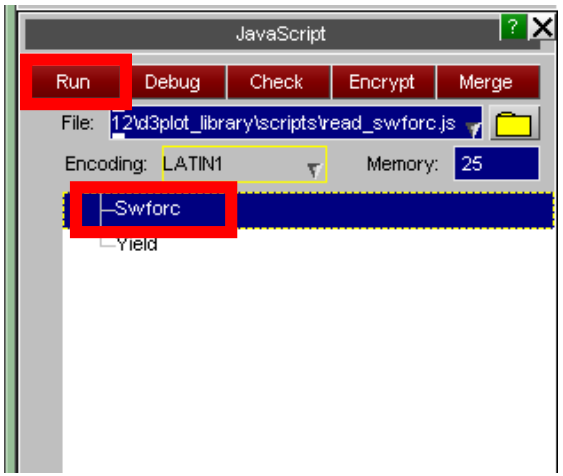
Primer Database/Templates

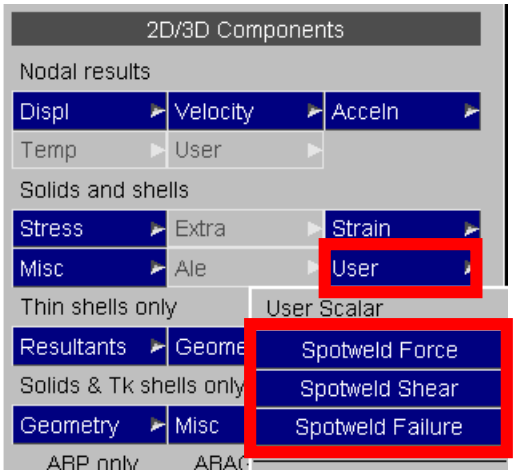


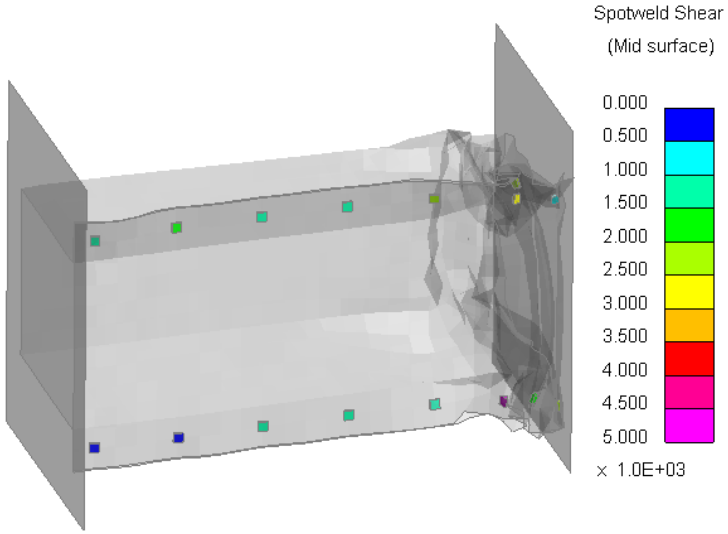
Assemble

Example – plotting spotweld forces

- This JavaScript is included with D3PLOT – **Swforc** listing in JavaScript menu.
- The script reads the *swforc* file, interpolates to the time-states in the d3plot file, and stores the data in user-defined data components.







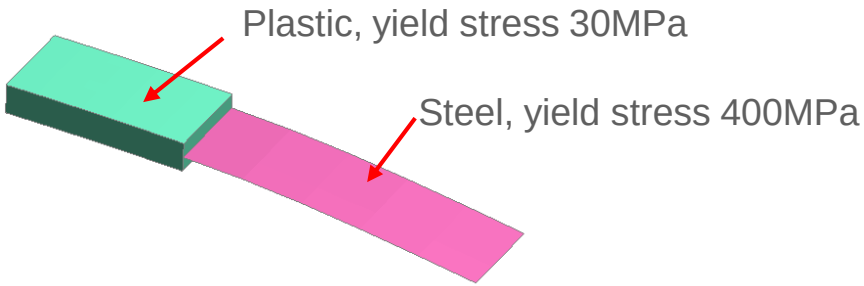
Run Script

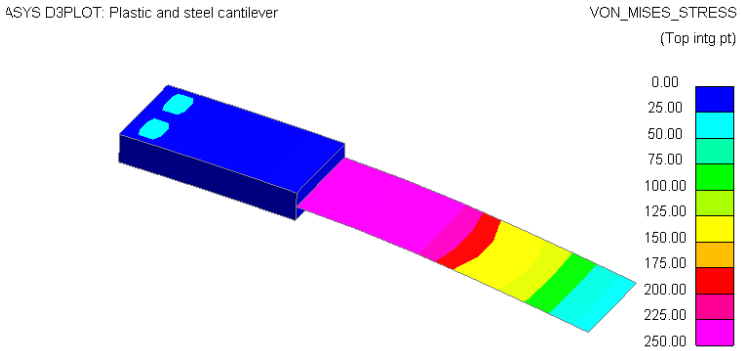
Select data component created by the script

Contour plot

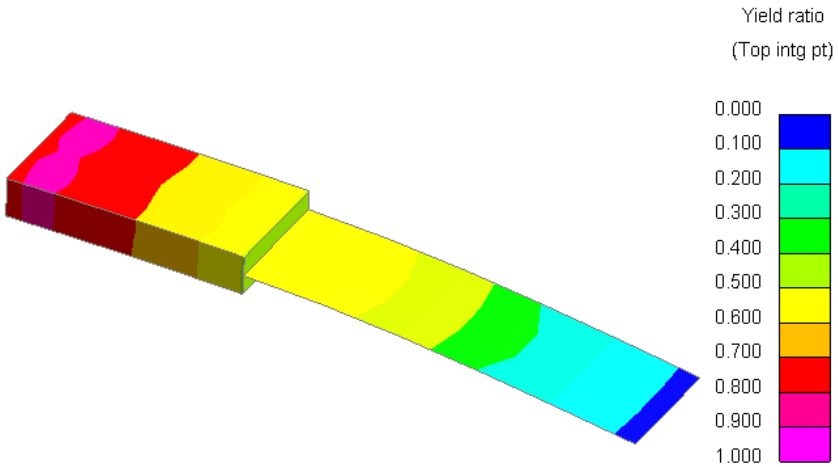
Example – how close to yield/failure?

This JavaScript is included in the *d3plot_library\examples* directory.





Stress is higher in the steel, but this does not tell us which material is closest to yield

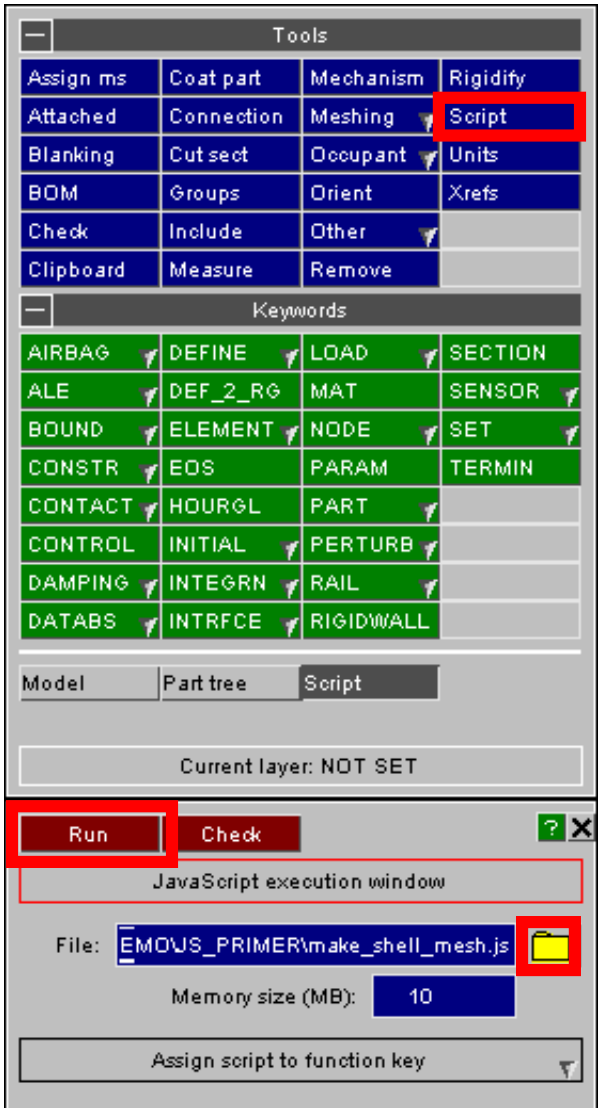


After running the script we can plot the ratio (Von Mises Stress)/(Yield Stress), using the correct yield stress for each material. Now we can see that the plastic part is closest to yield.

PRIMER JavaScript – Part 1

Running an existing script

- Start PRIMER. Do not read in a model.
- We are going to run a script that creates a model containing a simple mesh.
- Tools => Script, browse for the script *make_shell_mesh.js*
- Press Run
- A shell mesh should appear.



- In a text editor, open the script *make_shell_mesh.js*
- At this stage, we will not attempt to explain the details of the script
- Look through the script, reading the comments (lines starting with //)
- Near the top of the file, some variables are given values.
  - Read the meaning of the variables in the comments then change some of the numbers.
- Save the script (from the text editor).
- In PRIMER, press Run again.
  - A new mesh should appear (in a second model).
  - Check that your changes to the script have had the correct effect.

- You will need:
  - Existing scripts (that we provide) to use as examples
  - PRIMER JavaScript Manual – this document describes the extensions that we have written
  - You may also want a JavaScript textbook describing the Core functions.
- Write your script in the text editor, a few lines at a time. Then save and try to run it. Once these lines of script work correctly, add some more lines.
- We provide example scripts in these directories:
  - $OASYS\primer_library\scripts
  - $OASYS\examples\primer_scripts

This directory is for scripts that should be available as in the tree listing in the Scripts menu

This directory is for other examples that you can follow when writing scripts

- Create a new file in a text editor, called *script1.js*
- Write the following:

```
//   My first script


Message("Hello World");
```

Comment

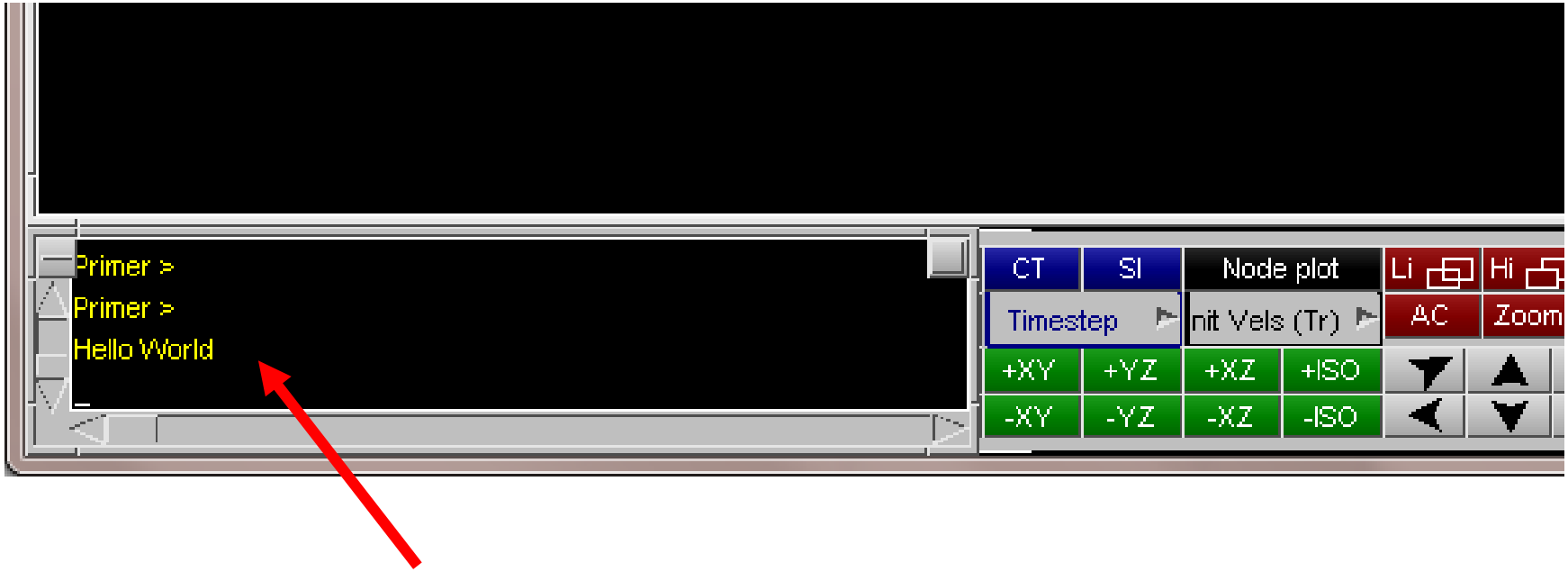
Blank lines are allowed; they help make the script easier to read

Lines end with a semicolon

Message is an extension function that we have provided, for printing messages in PRIMER's dialog box. Find the description in the PRIMER JavaScript manual.

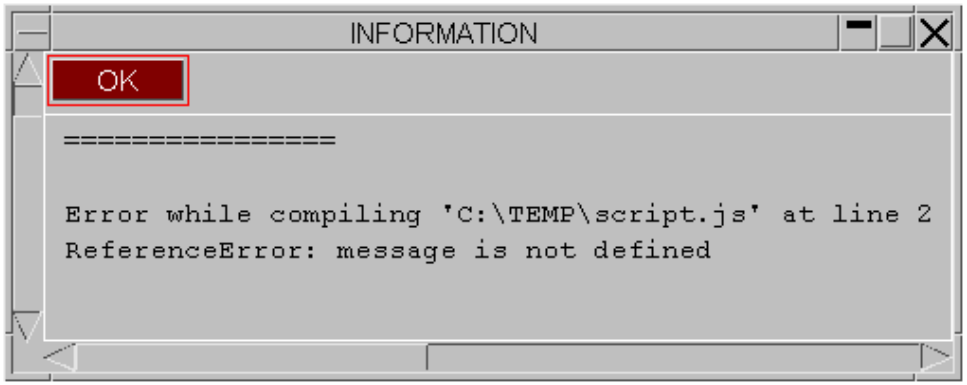
Arguments to functions (data provided as input or output) go in round brackets

- Run the script and look in PRIMER's dialog box. You may need to expand the dialog box or scroll through it.



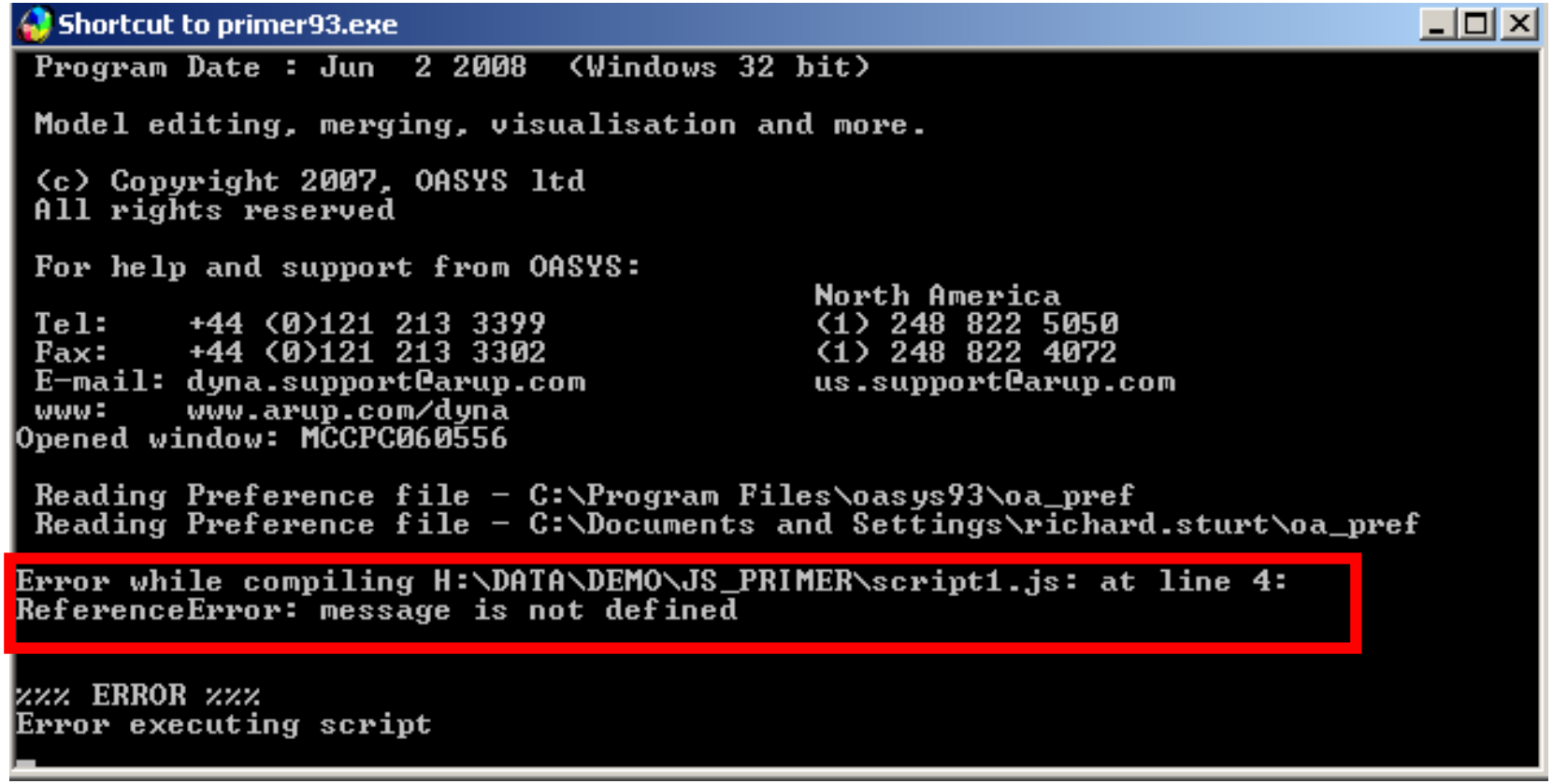
Text written using the Message function

- A typical error is to mis-spell a function name.
  - In your script, change *Message* to *message* (with lower case m). Save the script and re-run it.
- A message appears giving an error and details of what has gone wrong



- A Error message also appear in the dialogue box.

- There is also an error message printed in the PRIMER's start-up window.
- The interpreter tells you what is wrong, and on which line the fault occurred.



- Comments are lines that are ignored by the interpreter. They are used to help make your script intelligible. Use plenty of comments!
- Comments may be written in several ways:
  - From double-slash // to the end of the line is a comment
  - Start with /* and end with */ - anything in between is a comment, even if it covers several lines.

```
//
//   This is a comment line
//
```

```
/*
*   A block of comments
*   These are comment lines
*/
```

```
a = b + 1;     /* comments can be added on same line as coding */
c = d + 1;     // another way to write comments
```

- Variables allow you to store and manipulate data in the script. Unlike some other programming languages variables in JavaScript are *untyped*. This means that a variable can hold an integer, a floating point number, a string or an object. e.g.

```
i = 10;     // variable i contains the integer 10
i = 10.1;   // variable i now contains the floating point number 10.1
i = "ten"; // variable i now contains a string
```

- Variables are declared by using var.

```
var pi = 3.1415927;
```

Create a new variable containing the number 3.1415927

```
var n = new Node(m, 100, 20, 40, 10);
```

Create a new Node object.

The interpreter knows that n is a Node object because that is what the Node constructor returns.

- We can use var to create a variable or object before it is used, or when it is first used. e.g.

```
var pi;
pi = 3.1415927;
```

- If *var* is omitted, the Interpreter automatically declares the variable. However, variables declared automatically are “global” while those declared with *var* in functions are local to the function in which they are declared (see sections on functions and variable scope later).
  To avoid any confusion always declare variables using var.
- It is harmless to use *var* more than once for the same variable.

- Strings are made by enclosing characters in single or double quotes (‘ or “)

```
var s = ‘Hello, world!’;
var s = “Hello, world!”;
```

- If you want to use ‘ in your string then use double quotes (“) and visa versa.

```
var s = “You can use ‘single quotes’ inside double quotes”;
```

- Strings can be concatenated by using +

```
var s = “Hello, ” + “world!”; // produces “Hello, world!”
```

- The length of the string is contained in the property *length*. e.g. for s above

```
s.length
```

- A \ in a string has a special meaning. Combined with the next character in the string it represents a character that could otherwise not be allowed in the string. e.g.

```
var s = “You can use \”double quotes\” inside double quotes”;
```

- Some common escape sequences

\t	tab
\n	newline
\”	double quote
\’	single quote
\\	backslash

- Note the backslash case. This is important for filenames on Windows. e.g. to refer to a file C:\temp\nodes.csv as a string you would need to do:

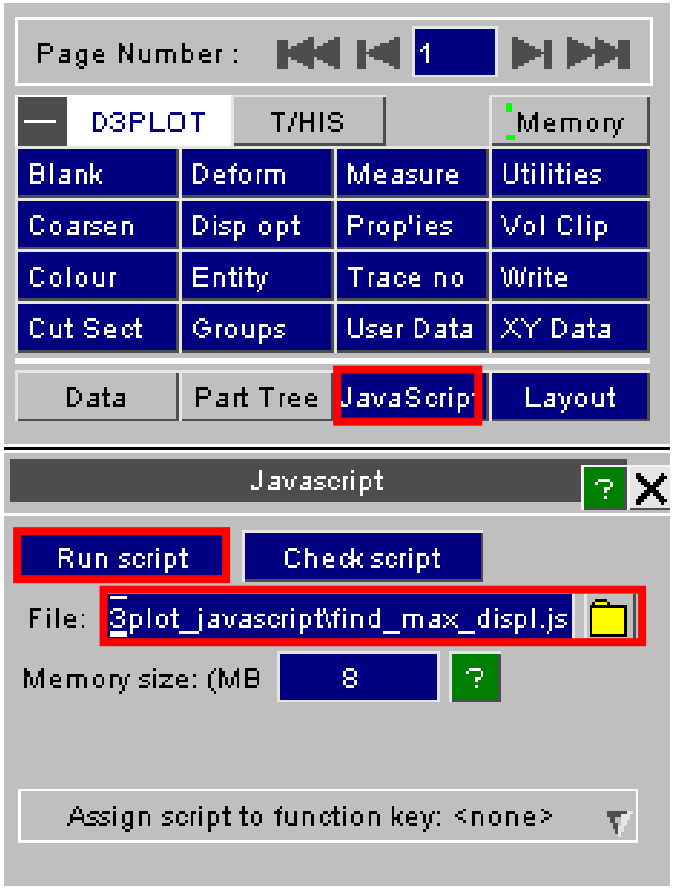
```
var s = “C:\\temp\\nodes.csv”;
```

D3PLOT JavaScript

Follow this link to skip this section of the course

Running an existing script

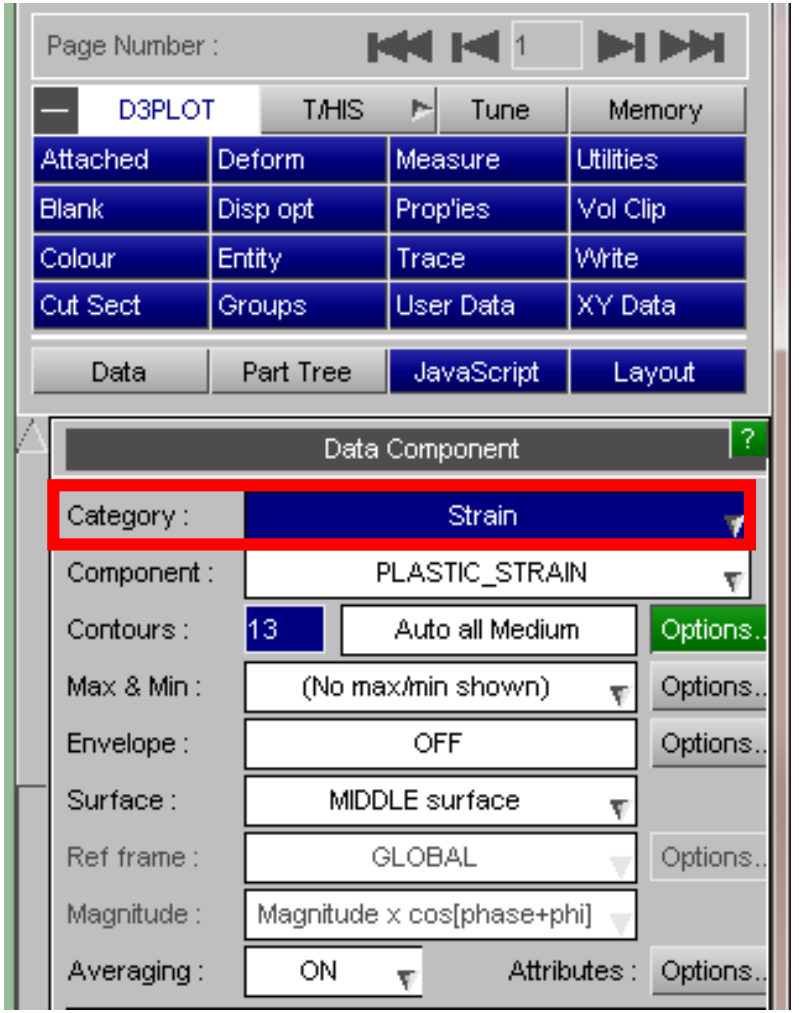
- Start D3PLOT, read the results file *base.ptf*
- We are going to run a script that checks all the output times, and reports the maximum displacement of any node at any time.
- Press JavaScript, browse for *find_max_displ.js*, press Run.
- The dialog box should show the result.



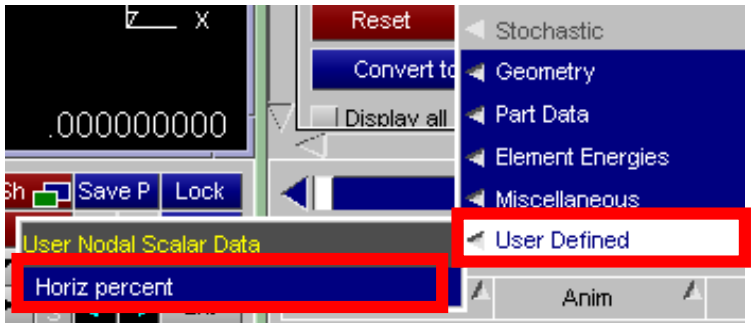

```
...processing state 107
...processing state 108
...processing state 109
...processing state 110
...processing state 111
...processing state 112
Maximum displacement = 342.10351338325O2
Javascript executed successfully
```

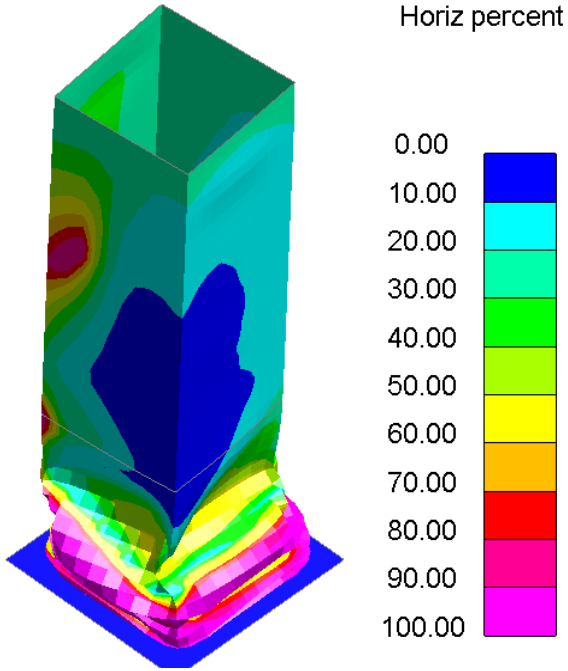
Running a script and plotting data

- We will now run a script that calculates, for each node at each timestate, the horizontal displacement (ignoring the vertical component), and expresses this as a percentage – i.e. when each node reaches its maximum horizontal displacement, its data value will be 100%.
- First, confirm that there are no pre-existing user-defined data components for nodes
  - Data Component menu,
  - Category drop down menu
  - User Defined selection



- Browse for the script *calc_horiz_displ_percent.js* and run it.
- This script creates a user-defined data component named “horiz percent”.
- In the Data Component menu, the User button is now live; select “Horiz percent”. Perform a shaded image plot (shortcut F).
- Animate the model.
- You could also try using XY_DATA to make a graph of the new data component versus time for a node.
- Note that a new file *base_1.ubd* has appeared in the directory with the model. This contains the data.
- Exit from D3PLOT, start D3PLOT again and read the same model. The new data is still available for plotting, without rerunning the script.





D3PLOT JavaScript or User-defined data?

- User-defined data components can be defined directly in D3PLOT, e.g. using the “simple formula” method. This is easier than writing a JavaScript to calculate results however it has limitations.
- When is a JavaScript needed?
  - “Simple formula” can be applied when the user-defined result for node n at time t depends only on existing data (e.g. displacements, velocities) for node n at time t. Similarly for elements – if the user-defined data at time t depends only on existing data for that element at time t.
  - If the user-defined data requires knowledge of results across multiple nodes/elements, or across multiple time-states, then “simple formula” cannot be used and a JavaScript is needed.
  - If the user-defined data is calculated using branching logic (i.e. is not a one-line mathematical formula), a JavaScript is needed.
  - If the user-defined data is calculated using data from an external file, in combination with the data in the results file, a JavaScript is needed.

- You will need:
  - Existing scripts (that we provide) to use as examples
  - D3PLOT Manual Appendix VI – this document describes the extensions that we have written
  - The extension functions that are common to D3PLOT and PRIMER are described in the Primer Javascript API manual.
  - You may also want a JavaScript textbook describing the Core functions.
- Write your script in the text editor, a few lines at a time. Then save and try to run it. Once these lines of script work correctly, add some more lines.
- We provide example scripts in these directories:
  - $OASYS\d3plot_library\scripts
  - $OASYS\d3plot_library\examples

This directory is for scripts that should be in the listings the Scripts menu

This directory is for other examples that you can follow when writing scripts

- Start your browser (or Acrobat) with the electronic D3PLOT manual.
- Go to Appendix VI.
- Look at the descriptions of the functions GetNumberOf, SetCurrentState, and GetData.
- Open the script *find_max_displ.js* in a text editor.
- Compare the descriptions and argument lists with the examples given in the script.

APPENDIX VI JAVASCRIPT INTERFACE

Description of functions and methods.

The following pages describe the functions ("methods") available from the Javascript interface.

Return value	Function name
boolean	CreateWindow(model_list);
boolean	DeleteWindow(window_list, (*dispose flag*))
boolean	SetWindowActive(window_id, active_flag)
integer	GetWindowMaxFrame(window_id)
boolean	SetWindowFrame(window_id, frame_number)
integer	GetWindowFrame(window_id)
object	GetWindowModels(window_id)
boolean	SetCurrentModel(model_id)
boolean	SetCurrentState(state_id)
integer	GetNumberOf(type, (*further args*))
boolean	QueryDataPresent(component, *(type)*)
double	GetTime(*(state)*)

- Arguments shown in brackets are optional.
- In the example below, GetNumberOf can have either 1 or 2 arguments.
- The first argument *type_code* is required (i.e. not in brackets).
- The second argument *state_id* is optional.

integer GetNumberOf(type_code, (*state_id*))

Returns the quantity ("number of") items of type <type_code> in the current model.

- Unlike the PRIMER Javascript interface (in which data is treated as objects and object properties), in D3PLOT data is accessed using integer indices (since D3PLOT stores data in arrays).
- For example, this script would write the labels of the first 3 nodes to the dialog box:

```
var lab1 = GetLabel(NODE, 1);
var lab2 = GetLabel(NODE, 2);
var lab3 = GetLabel(NODE, 3);
Message("Label of first node = " + lab1);
Message("Label of second node = " + lab2);
Message("Label of third node = " + lab3);
```

- Watch out: the index for the first node (or element, etc) is 1, NOT zero.

- In D3PLOT, only one model may be “current” at any point in the script, allowing its data to be accessed.
- If more than one model is present, first set the current model before accessing data (SetCurrentModel command).
- Likewise, only one time-state may be “current” at any point in the script.
- Before accessing results data, set the current state (SetCurrentState command).

```
SetCurrentModel(1);                 // Assume we are working with Model 1

SetCurrentState(10);                // Set the 10-th time-state to be current
var dx10 = GetData(DX, NODE, 1); // Get the X-displacement of the first node

SetCurrentState(15);                // Set the 15-th time-state to be current
var dx15 = GetData(DX, NODE, 1); // Get the X-displacement of the first node
```

The GetData function

- GetData is used to access all results, and also some input data such as original (basic) coordinates.

```
var dx10 = GetData(DX, NODE, 1); // Get the X-displacement of the first node
```

Constants are used to identify the data component. To see a full list of these, follow the link

double *or* **double array GetData(component, type_code, item,** *(int_pnt), (extra), (fr_of_ref), (state_id), (dda)***)**

Returns the data for <component> of <item> of type <type_code>.

The return value is scalar, array[3] or array[6] for scalar, vector and tensor components respectively.

WARNING: If the function arguments are grammatically correct but the requested data component is not present in the database, then 1, 3 or 6 zeros are returned as required, ***and no warning n***
function QueryDataPresent() to check that an optional data component is actually present in a database before attempting to extract its values.

Arguments:	**<component>**	**Constant**	A valid component code (eg **DX, SXY**)	Only valid codes in the list below are permitted.
	<type_code>	**Constant**	A valid element type code (**SOLID**, etc)	The type of the item
	<item>	**Integer**	If +ve: The internal item number starting from 1 If -ve: The external label of the item	Internal item numbers will be many times faster to proce
	<int_pnt>	**Integer**	**Optional**: • If +ve is an integration point id (1 = lowest), • Alternatively one of the codes **TOP, MIDDLE, BOTTOM** Use zero to define a null "padding" argument	Integration points are only meaningful for some compo stresses. This argument may be omitted if not needed.
	<extra>	**Integer**	**Optional**:	This argument is only necessary for a few components

- GetData is used to access all results, and also some input data such as original (basic) coordinates.

```
var dx10 = GetData(DX, NODE, 1); // Get the X-displacement of the first node
```

Constants are used to identify the entity type. To see a full list of these, follow the link

double *or* **double array** GetData(component, type_code, item, *(int_pnt), (extra), (fr_of_ref), (state_id), (dda)*)

Returns the data for <component> of <item> of type <type_code>.

The return value is scalar, array[3] or array[6] for scalar, vector and tensor components respectively.

WARNING: If the function arguments are grammatically correct but the requested data component is not present in the database, then 1, 3 or 6 zeros are returned as required, ***and no warning n***
function QueryDataPresent() to check that an optional data component is actually present in a database before attempting to extract its values.

Arguments:	**<component>**	**Constant**	A valid component code (eg **DX, SXY**)	Only valid codes in the list below are permitted.
	<type_code>	**Constant**	A valid element type code (**SOLID**, etc)	The type of the item
	<item>	**Integer**	If +ve: The internal item number starting from 1 If -ve: The external label of the item	Internal item numbers will be many times faster to proce
	<int_pnt>	**Integer**	**Optional**: • If +ve is an integration point id (1 = lowest), • Alternatively one of the codes **TOP, MIDDLE, BOTTOM** Use zero to define a null "padding" argument	Integration points are only meaningful for some compo stresses. This argument may be omitted if not needed.
	<extra>	**Integer**	**Optional**:	This argument is only necessary for a few components

The GetData function

- When using GetData to access shell element stresses, the integration point must be specified.
- Can use 1,2,3… or TOP, MIDDLE, BOTTOM.

```
var stress = GetData(SXX, SHELL, 1, TOP); // Get the X-stress of the first shell
```

double *or* **double array GetData(component, type_code, item,** *(int_pnt)*, *(extra)*, *(fr_of_ref)*, *(state_id)*, *(dda)*)

Returns the data for <component> of <item> of type <type_code>.

The return value is scalar, array[3] or array[6] for scalar, vector and tensor components respectively.

WARNING: If the function arguments are grammatically correct but the requested data component is not present in the database, then 1, 3 or 6 zeros are returned as required, ***and no warning n***
function QueryDataPresent() to check that an optional data component is actually present in a database before attempting to extract its values.

Arguments:	**<component>**	**Constant**	A valid component code (eg **DX**, **SXY**)	Only valid codes in the list below are permitted.
	<type_code>	**Constant**	A valid element type code (**SOLID**, etc)	The type of the item
	<item>	**Integer**	If +ve: The internal item number starting from 1 If -ve: The external label of the item	Internal item numbers will be many times faster to proce
	<int_pnt>	**Integer**	**Optional**: • If +ve is an integration point id (1 = lowest), • Alternatively one of the codes **TOP**, **MIDDLE**, **BOTTOM**	Integration points are only meaningful for some compo stresses. This argument may be omitted if not needed.
			Use zero to define a null "padding" argument	
	<extra>	**Integer**	**Optional**:	This argument is only necessary for a few components

- First use GetNumberOf to find how many nodes, time-states, etc. there are; then use a *for* loop.
- The script will run more quickly if done in this order:
  - Loop through the time-states
  - For each state, process data for all nodes/elements/etc.

```
var nstate = GetNumberOf(STATE);    // Get number of time-states
var nnode  = GetNumberOf(NODE);     // Find number of nodes in model

for (istate=1; istate<=nstate; istate++)
{
   SetCurrentState(istate);
   for (j=1; j<=nnode; j++)
   {
      do something…
   }
}
```

- The script will run more slowly if done in this order:
  - Loop through nodes/elements etc
  - For each node/element, process data for all time-states

```
var nstate = GetNumberOf(STATE);    // Get number of time-states
var nnode  = GetNumberOf(NODE);     // Find number of nodes in model

for (j=1; j<=nnode; j++)
{
   for (istate=1; istate<=nstate; istate++)
   {
      SetCurrentState(istate);
      do something…
   }
}
```

Not recommended – will be slower

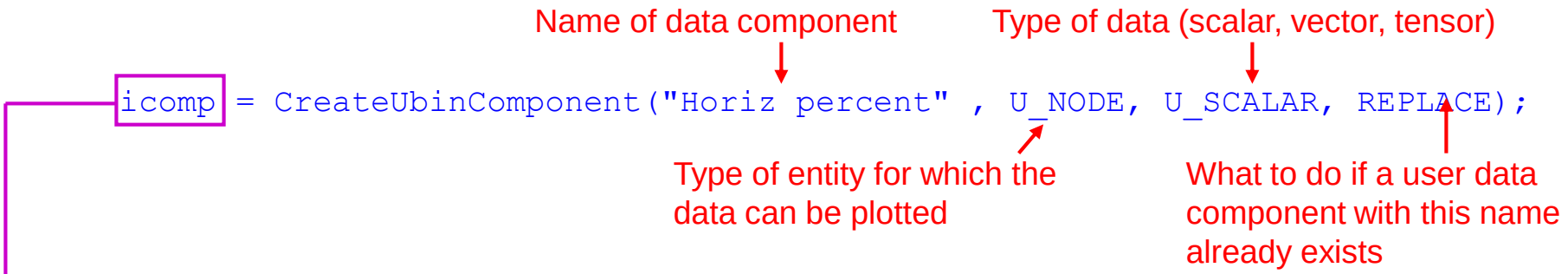
- If there is a good reason to write the loops in this way (e.g. if complex operations are to be performed on a complete time-history for each node or element), consider using the direct disk access method (DDA argument in the function GetData).

Exercise - Accessing data from a model

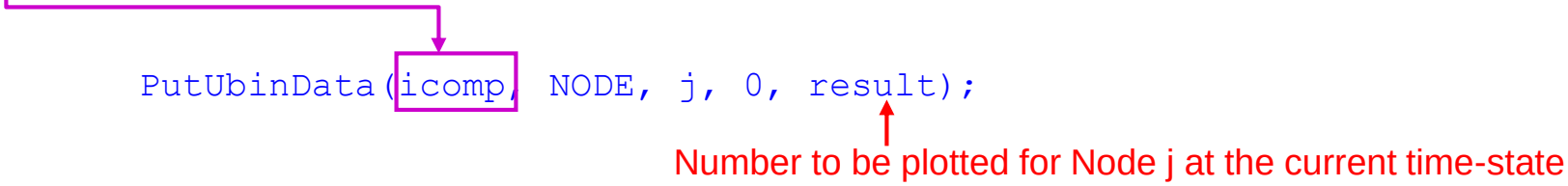
- Start from *find_max_displ.js,* and save the script under a new name *access_data_1.js*.
- Edit the script where needed to do the following:
  - Loop through all the time-states
  - Find the maximum von Mises stress in any shell element at each time-state
    - Read the notes in the manual for the GetData function regarding surface selection (data for shells is present at top, middle, and bottom integration points).
  - Write the maximum value to the dialog box (for each time-state)
- The result should be 864.97

In case of problems with this exercise, look at access_data_d3plot_1_complete.js

- It is often useful to calculate data for each node (or element), and display it as a contour plot.
- This is done using User Binary (UBIN) Data Components.
- First, create a data component using CreateUbinComponent. This function returns a “handle” (constant) by which the data component may be referenced:



- Data may then be stored using PutUbinData:



- Scalar data = one value per node or element.
- This data may be viewed by contour plotting, and is also available from WRITE and XY_DATA
- Any number of data components can be in use within the script simultaneously, e.g.

```
icomp1 = CreateUbinComponent("My data 1" , U_NODE, U_SCALAR, REPLACE);
icomp2 = CreateUbinComponent("My data 2" , U_NODE, U_SCALAR, REPLACE);

for (istate=1; istate<=nstate; istate++)
{
   SetCurrentState(istate);
   for (j=1; j<=nnode; j++)
   {
      … (calculate) …
      PutUbinData(icomp1, NODE, j, 0, result1);
      PutUbinData(icomp2, NODE, j, 0, result2);
   }
}
```

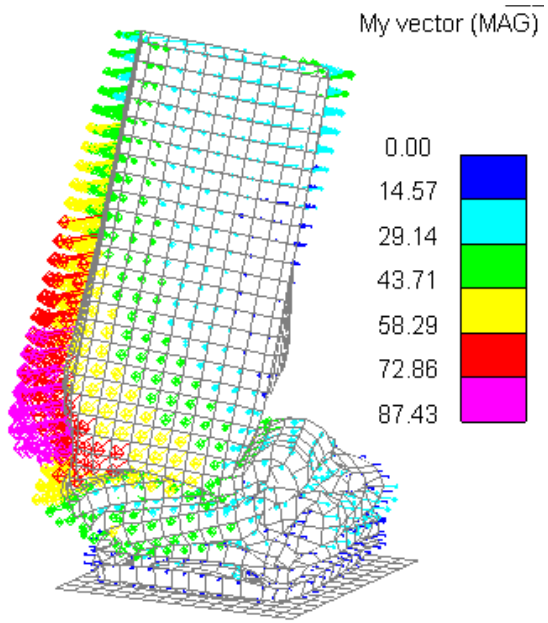
- Start from *access_data_d3plot_2_incomplete.js*, save under a new name *access_data_d3plot_2.js*
- Add a statement to create a new user data component “failure index”. This will be a scalar variable for shell elements.
- The coding to calculate the failure index has already been written. This is what it is supposed to do:
  - If plastic strain = 0 (i.e. the element has not yielded)
    - failure index = (von Mises stress)/(yield stress), where yield stress = 200MPa
  - If plastic strain > 0
    - failure index = 1.0 + (plastic strain)/(failure strain), where failure strain = 0.8.
  - Thus the index will rise from zero to 1.0 at yield, and to 2.0 when the failure strain is reached. Note, however, that the input data did not include any failure strain so the elements pass the “failure strain” without failing. Find maximum index at top, middle and bottom surfaces.
- Add a statement to write the calculated failure index to the user data component.

In case of problems with this exercise, look at access_data_d3plot_2_complete.js

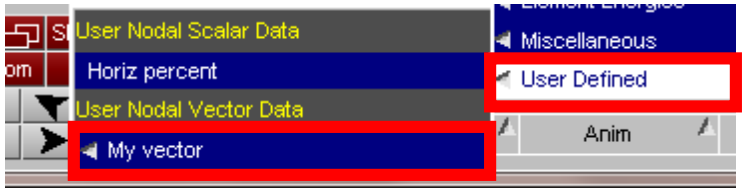
- Vector data = 3 values per node
- Store an array instead of a single value

Type of data = vector

```
icomp = CreateUbinComponent("My vector" , U_NODE, U_VECTOR, REPLACE);
var my_result = new Array();
for (j=1; j<=nnode; j++)
{
   my_result[0] = GetData(DX, NODE, j);
   my_result[1] = GetData(DY, NODE, j);
   my_result[2] = 0.0;
   PutUbinData(icomp, NODE, j, 0, my_result);
}
```



- The data may be plotted as arrows
- Similar process for creating
  - Beam forces (6 numbers)
  - Shell/solid tensor (6 numbers)



- Any D3PLOT capability that can be accessed via command line (commands typed in the dialog box, or written in a command file) can be issued from a JavaScript.
  - Use the functions DialogueInput or DialogueInputNoEcho:

```
DialogueInput("/BLANK ALL","UNBLANK PART 1","REDRAW");
```

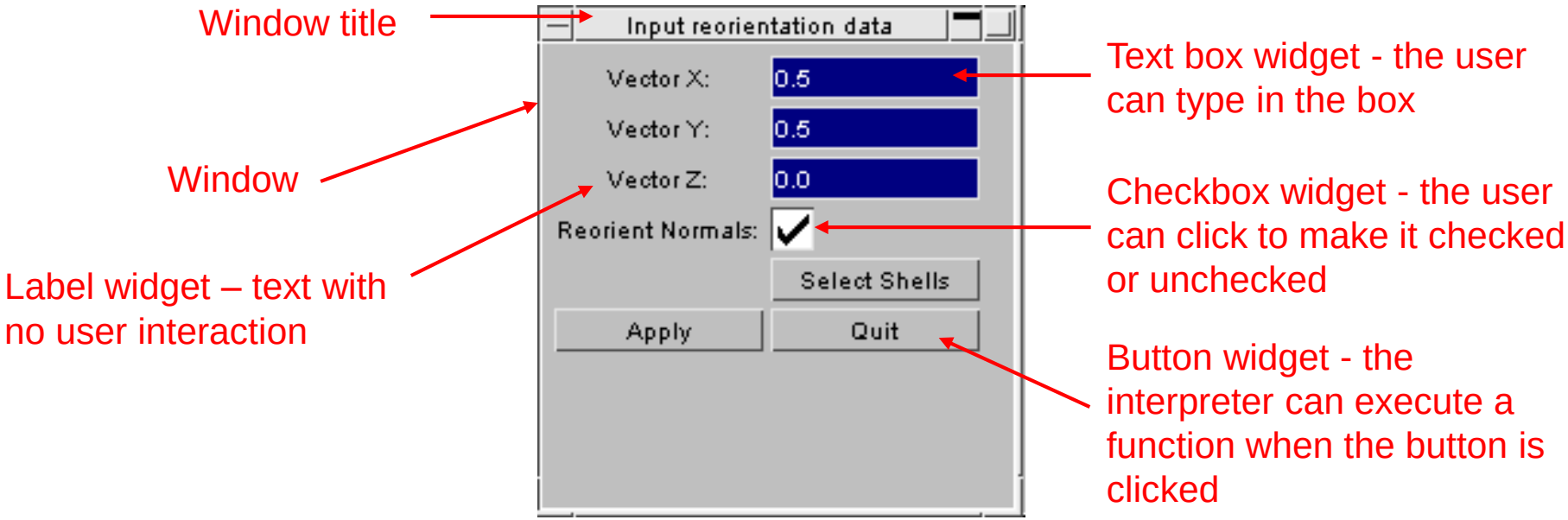
- After each call to DialogueInput (or DialogueInputNoEcho), D3PLOT returns automatically to the main menu.
- If you need to issue a sequence of commands without returning to the main menu, use a single call with multiple arguments (separated by commas), as in the above example.

- Some ready-made windows are available in D3PLOT:
  - See the PRIMER JavaScript manual in the Window Class.
  - All of the Window and Widget class functions from PRIMER are also available in D3PLOT. For example:

Window.Information

- This subject is covered later in the course

- New menus may be created using the Window and Widget classes.
  - “Window” = the window containing a floating menu
  - “Widget” = buttons, text boxes, text labels, checkboxes, etc
- See PRIMER Javascript manual for details.
- The functions and techniques are identical in D3PLOT and PRIMER

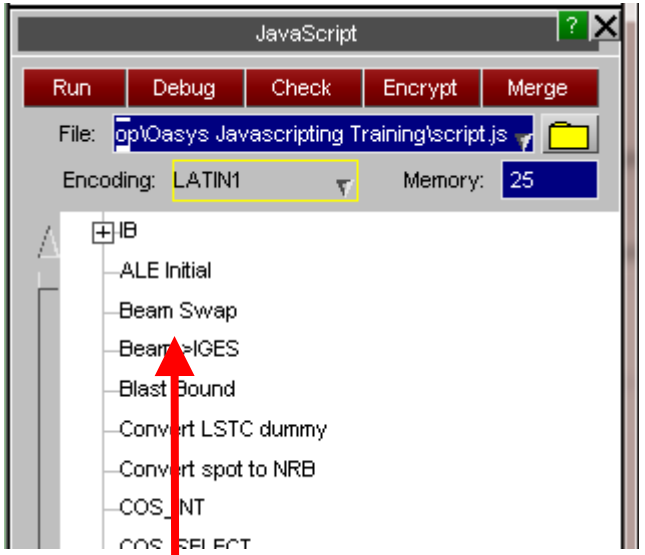


- Commonality between D3PLOT and PRIMER Javascripting
  - Core functions are (of course) identical
  - Extension functions for creating GUIs and reading/writing external files are identical in PRIMER and D3PLOT
  - Functions for accessing data are different, reflecting the different internal structure of the two programs.
    - PRIMER JavaScripts are object-oriented,
    - While D3PLOT JavaScripts are mostly not object-oriented.

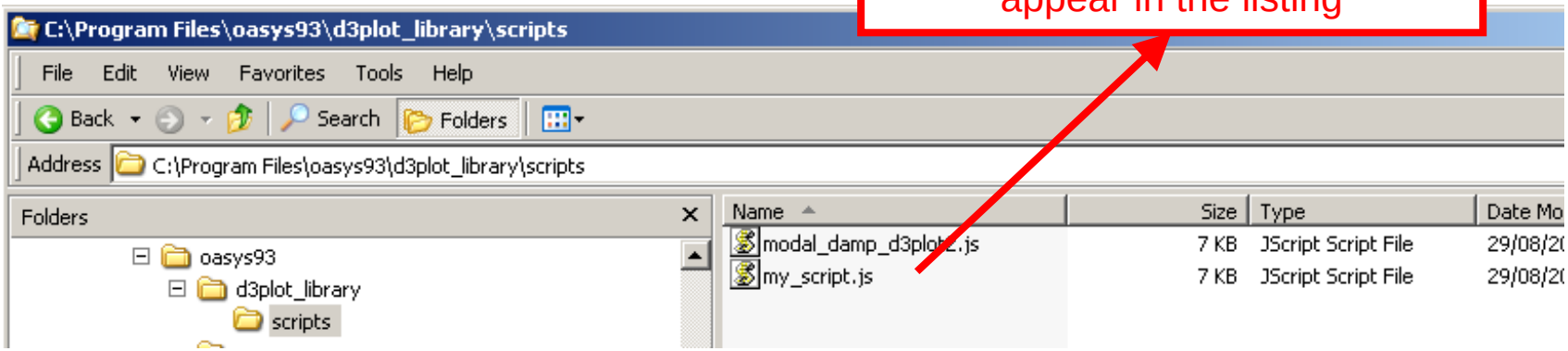
PRIMER JavaScript – Part 2

Making your scripts available to others

- Any script can be run by browsing for the script file.
- To make a script available more easily, copy it into the directory
  - $OASYS\d3plot_library\scripts or
  - $OASYS\primer_library\scripts.
  - For each script in this directory, an entry appears in listing in the Script menu.

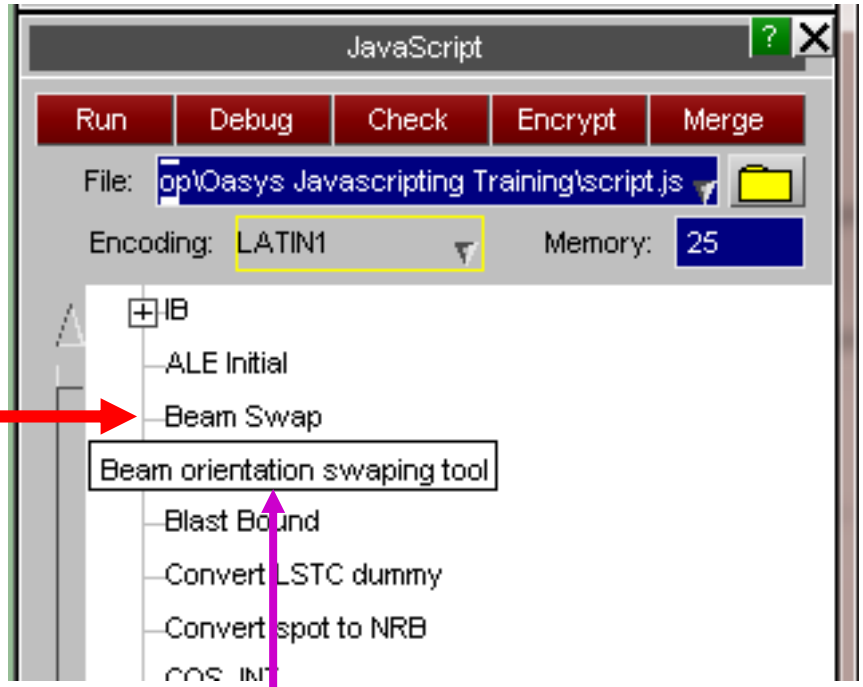


By default, the script filenames appear in the listing



- To change the name of the button (and optionally to add hover text for the button) add a special comment at the top of the file containing text:

  name:<name for button>
  description:<hover text to display>




```
// name: Beam Swap
// description: Beam orientation swapping tool
//
//    It is assumed that Model 1 is the d3eigv file
//    The script writes out damping data per mode
//
```

Oasys JavaScript extensions

- Extensions to core JavaScript are written by the Oasys software development team for interaction with PRIMER's model data. These extensions can be called in users' scripts.
- To access each type of LS-DYNA keyword requires a new class to be added to JavaScript by Oasys.
- Examples of classes available to access keywords are:

```
*BOUNDARY_SPC, _PRESCRIBED_MOTION
*CONSTRAINED_NODAL_RIGID_BODY
*CONTACT
*DEFINE_CURVE, _VECTOR
*ELEMENT_BEAM,   _SOLID,   _SHELL,   _DISCRETE,   _MASS
*INITIAL_STRESS_SHELL
*LOAD_NODE
*NODE
*PART, *MAT, *SECTION, *HOURGLASS
*SET
```

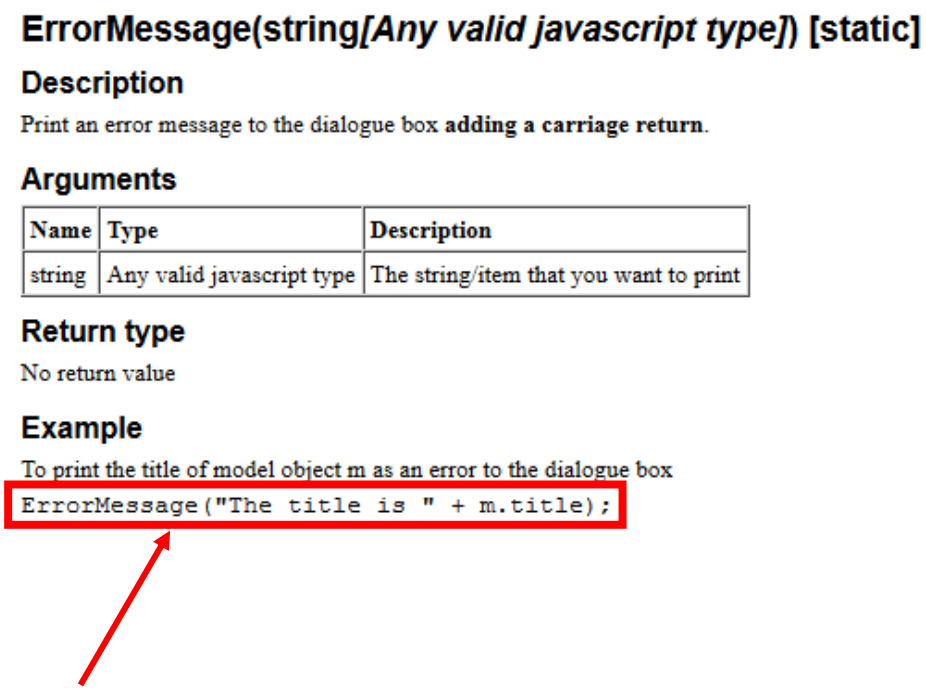
- These PRIMER capabilities can also be accessed via Oasys extensions:

```
All command-line commands
Blanking
Colour
Connections (spotwelds etc)
Image (write JPG etc)
Merge Nodes
Remove
View, redraw, etc
Xrefs
+others
```

- Menus can be created
- Functions to read and write external files (including XML files) are also provided as extensions, since these are not part of Core JavaScript.

Using the PRIMER JavaScript manual

- To find what functions are available for writing messages, open the PRIMER JavaScript manual.
- Follow the link to the global class. This class contains stand-alone functions – more on this later.
- Look at the detailed description for the function ***ErrorMessage*.**
- Use this function in your script to write an error message
  - "example error message".



ErrorMessage(string*[Any valid javascript type]*) [static]

Description

Print an error message to the dialogue box **adding a carriage return**.

Arguments

Name	Type	Description
string	Any valid javascript type	The string/item that you want to print

Return type

No return value

Example

To print the title of model object m as an error to the dialogue box

```
ErrorMessage("The title is " + m.title);
```

Use the example in the manual entry to help write your own error message.

You can omit "+ m.title", this will be explained later.

In case of problems with this exercise, look at script1_complete.js

- A class is a grouping to which functions and data can belong.
  - For example, each keyword supported by the JavaScript interface has its own class (e.g. Node, Shell, Part…). There are also classes for external Files, GUI generation, and other items.
- For each class you can then create individual *objects* that are *instances* of that class.
  - For example, the Node class represents *NODEs in PRIMER. A Node *object* represents a single *NODE entity and is an *instance* of the Node class.
- An object has properties that depend on its class.
  - A Node object has properties that include x-coordinate, y-coordinate, label, etc.
- To find the properties of each type of object, look in the PRIMER JavaScript manual. The manual is organised into classes.
  - To find the properties of a Node object, look in the Node class.

Oasys PRIMER

Node properties

Name	Type	Description
exists	logical	true if node exists, false if referred to but not defined. (read only)
ndof	integer	Number of degrees of freedom (SCALAR and SCALAR_VALUE only).
nid	integer	Node number
rc	integer	Rotational constraint (0-7)
scalar	integer	The type of the node. Can be false (*NODE), Node.SCALAR (*NODE_SCALAR) or Node.SCALAR_VALUE (*NODE_SCALAR_VALUE)
tc	integer	Translational constraint (0-7)
x	float	X coordinate
x1	integer	Initial value of 1st degree of freedom (SCALAR_VALUE only).
x2	integer	Initial value of 2nd degree of freedom (SCALAR_VALUE only).
x3	integer	Initial value of 3rd degree of freedom (SCALAR_VALUE only).
y	float	Y coordinate
z	float	Z coordinate

If n is a node object, then n.x is its x-coordinate. For example, to copy the x-coordinate of this node to a new variable xvalue:

var xvalue = n.x; Note the . (full stop) between the object and the property name.

Node class

The Node class gives you access to node cards in PRIMER. More...

Class functions

- BlankAll(Model *[Model]*, redraw (optional) *[boolean]*)
- BlankFlagged(Model *[Model]*, flag *[Flag]*, redraw (optional) *[boolean]*)
- Create(Model *[Model]*, modal (optional) *[boolean]*)
- First(Model *[Model]*)
- FirstFreeLabel(Model *[Model]*, layer (optional) *[Include number]*)
- FlagAll(Model *[Model]*, flag *[Flag]*)
- ForEach(Model *[Model]*, func *[function]*, extra (optional) *[any]*)
- GetAll(Model *[Model]*)
- GetFromID(Model *[Model]*, number *[integer]*)
- Last(Model *[Model]*)
- LastFreeLabel(Model *[Model]*, layer (optional) *[Include number]*)
- Merge(Model *[Model]*, flag *[Flag]*, dist *[float]*, label (optional) *[integer]*, position (optional) *[integer]*)
- NextFreeLabel(Model *[Model]*, layer (optional) *[Include number]*)
- Pick(prompt *[string]*, Model (optional) *[Model]*, modal (optional) *[boolean]*, button text (optional) *[string]*)
- RenumberAll(Model *[Model]*, start *[integer]*)
- RenumberFlagged(Model *[Model]*, flag *[Flag]*, start *[integer]*)
- Select(flag *[Flag]*, prompt *[string]*, limit (optional) *[Model or Flag]*, modal (optional) *[boolean]*)
- Total(Model *[Model]*, exists (optional) *[boolean]*)
- UnblankAll(Model *[Model]*, redraw (optional) *[boolean]*)
- UnblankFlagged(Model *[Model]*, flag *[Flag]*, redraw (optional) *[boolean]*)
- UnflagAll(Model *[Model]*, flag *[Flag]*)
- UnsketchAll(Model *[Model]*, redraw (optional) *[boolean]*)

Member functions

- Blank()
- Blanked()
- ClearFlag(flag *[Flag]*)
- Edit(modal (optional) *[boolean]*)
- Error(message *[string]*, details (optional) *[string]*)
- Flagged(flag *[Flag]*)
- GetAttachedShells(recursive (optional) *[boolean]*)
- GetFreeEdgeNodes()
- GetInitialVelocities()
- GetParameter(prop *[node property]*)
- Keyword()
- KeywordCards()
- Next()
- NodalMass()

Class functions “hang off” the Class, e.g.

```
var n = Node.First (m);
```

Class name

Member functions “hang off” an individual object of this class, e.g.

```
var n2 = n.Next();
```

n must be a Node object

First(Model[*Model*]) [static]

Description

Returns the first node in the model.

Arguments

Name	Type	Description
Model	Model	Model to get first node in

Return type

Node object (or null if there are no nodes in the model).

Example

To get the first node in model m:

```
var n = Node.First(m);
```

The first (and only) argument must be a model object.

The output from this function is a Node object

This line will create a Node object containing the data for the first node in the model.

Class name (needed because *First* is a Class function)

Edit(modal (optional)*[boolean]*)

Description

Starts an interactive editing panel to edit the node.

Arguments

Name	Type	Description
modal (optional)	boolean	If this window is modal (blocks the user from doing anything else in PRIMER until this window is dismissed). If omitted the window will be modal.

Optional arguments may be omitted. If there are any compulsory arguments, they go before the optional arguments.

Return type

No return value

This function has no return value. The initial `var` could be omitted.

Example

To edit node n:

```
var n.Edit();
```

Even if a function has no arguments, it still needs brackets

- Global function without a return value:

```
Message("Hello World");
```

- Global function with a return value:

```
var iflag = AllocateFlag();
```

- Class function without a return value:

```
Node.BlankAll(m);
```

- Class function with a return value:

```
Node.BlankAll(m);
```

- Member function without a return value:

```
n.Edit(m);
```

- Member function with a return value:

```
n = n.Next();
```

- Copy the example from the Primer Javascript manual, paste it into your script:.

Example

To pick a part from model m giving the prompt 'Pick part from screen':

```
var p = Part.Pick('Pick part from screen', m);
```

```
var p = Part.Pick("Pick part from screen",m);
```

- If necessary, change the variable names to match those in your script.

```
var p1 = Part.Pick("Pick part from screen",m1);
```

Exercise - Accessing data from a model

- In this example, we will find and print the label of the first node in the model.
- Start from the script named *access_data.js*
- Get model 1 as a Model Object (using GetFromID in the Model Class)
- Get the first node from this model as a Node Object (use First in the Node class)
- Message is used to write the node's ID to the dialog box

```
//   Get Model 1 as a Model Object m
(add your code here)

//   Get the first node as a node object n
(add your code here)

//   Write the node's label (n.nid) to dialog box
Message("The label of the first node is " + n.nid);
```

Delete these lines, add your code instead

Note how the text of the message is built up from text and numbers, separated by +

- Read the model *shell_mesh1.key* into PRIMER
- Run the script, check in the dialog box that the correct message is displayed.

- Add more lines to your script as follows:
- Use the function *GetFromID* to create a Node object n2 containing the data for Node 1020.
- Use the function *Println* to write the coordinates of this node to the Start-up window.
  - (The function Println is in the Global class – look it up in the manual).
- Save and run the script, check the result.

In case of problems with this exercise, look at access_data_1_complete.js

Note on writing text with numbers

Script:

```
Println("Coordinates of node 1020 = " + n2.x + n2.y + n2.z);
```

Result:

```
Coordinates of node 1020 = 1207.50
```

No spaces or commas were placed between the numbers – the result is impossible to understand

Script:

```
Println("Coordinates of node 1020 = " + n2.x + ", " + n2.y + ", " + n2.z);
```

Result:

```
Coordinates of node 1020 = 120, 7.5, 0
```

Spaces and commas were placed between the numbers – the result can be understood

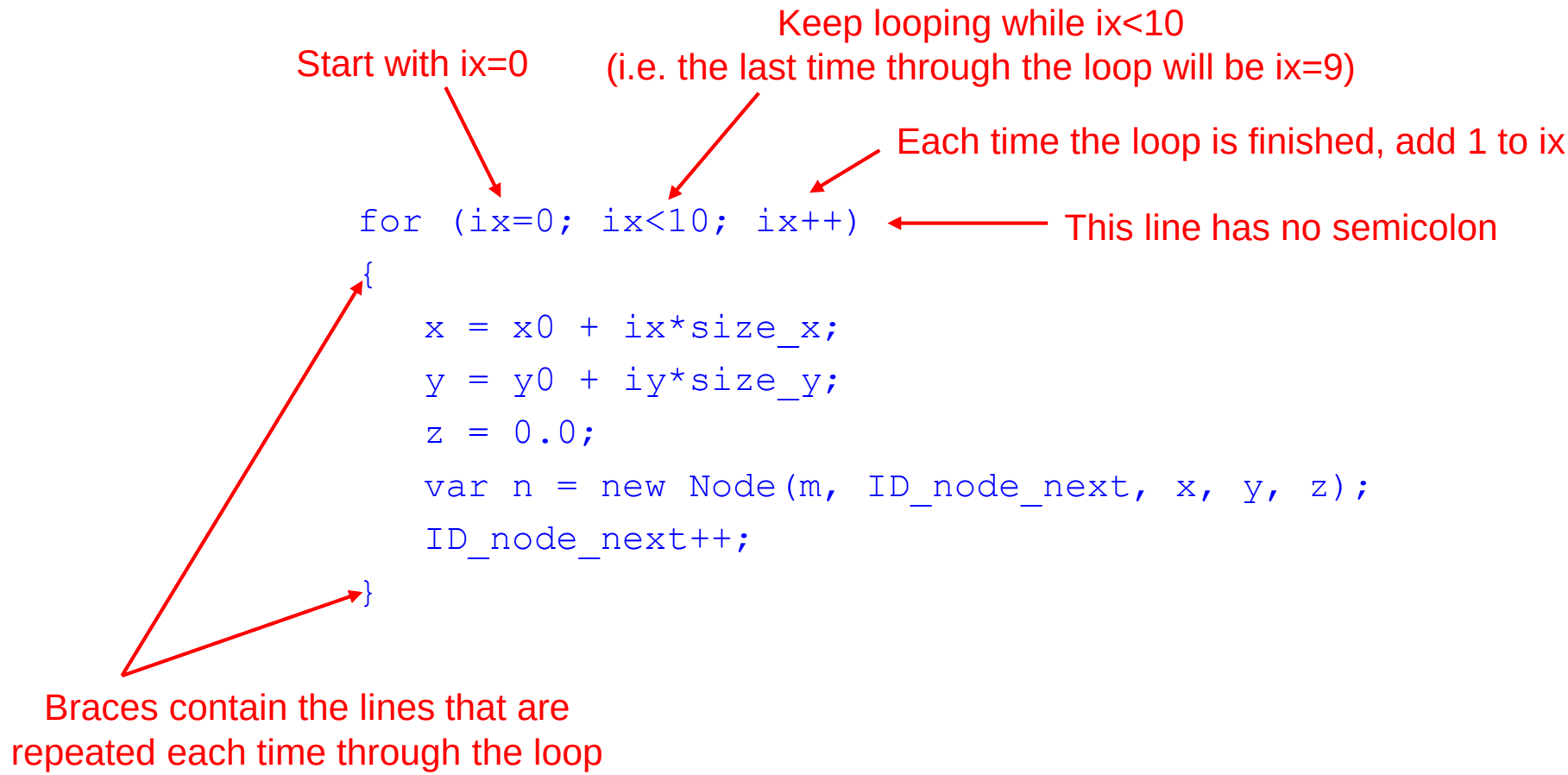
- We will now change the coordinates of node 1020.
- Add this line to the end of your script:

```
n2.x = 120.0;
```

- Save and run the script. You will have to re-draw the model in different modes (SH then LI) to see the result.
- Note that the data in the JavaScript (e.g. ***n2.x***) is a “mirror” of the actual data in PRIMER. When the properties of a node object are changed in the script, the equivalent data in the model changes in PRIMER automatically.
- Add the function *UpdateGraphics* to the end of your script (this function is in the Model class).
- Delete the model from PRIMER, re-read *shell_mesh1.key*.
- Run the script. Now, you should see the image update immediately.

In case of problems with this exercise, look at access_data_2_complete.js

- This is an example of a loop using the ***for*** command:



- **“for”** loops can be nested:

```
for (iy=0; iy<9; iy++)
{
    for (ix=0; ix<9; ix++)
    {
        x = x0 + ix*size_x;
        y = y0 + iy*size_y;
        z = 0.0;
        var n = new Node(m, ID_node_next, x, y, z);
        ID_node_next++;
    }
}
```

Each loop has a different counter (ix and iy)

- The indenting of the script lines in the loop is not compulsory but makes the script easier to read.

- **“while”** means “Keep looping while the condition remains true”:

```
ix = 0;
while (ix<10)
{
   x = x0 + ix*size_x;
   y = y0 + iy*size_y;
   z = 0.0;
   var n = new Node(m, ID_node_next, x, y, z);
   ID_node_next++;
   ix++;
}
```

The condition

- Example of a single statement dependent on "if":

```
if (x<0.0) Message("x is negative");
```

semicolon

- Example of multiple statements dependent on "if"

```
if (x<0.0)
{
    Message("x is negative");
    y = y + 10.0;
}
```

no semicolon

braces

- Example of branching "if":

```
if (x<0.0)
{
   Message("x is negative");
   y = y + 10.0;
}
else if (y<0.0)
{
   Message("y is negative");
   z = z + 10.0;
}
```

- More logical tests:

```
if (x==0.0)                 = if x is equal to 0.0 NOTE == not =
if (x!=0.0)                 = if x is not equal to 0.0
if (x==0.0 && y==0.0)       = if x=0 and y=0
if (x==0.0 || y==0.0)       = if x=0 or y=0
if (x)                      = if x exists and has a non-zero value
if (!(x==4.0))              = if the condition in brackets is false
if (!x)                     = if x does not exist or is zero
```

Example – looping through nodes

```
var n = Node.First(m);
while (n)
{
    Message ("Node label = " + n.nid);
    n = n.Next();
}
```

This means, keep repeating the lines within the braces while n has a value (or, until n becomes “null”).

Could be any code here that does something with a node

The last step in the loop is to replace n with the next node after the old node n.

If n is the last node, then n.Next will be null.

- We are now going to change the script so that it can fold the mesh.
- Comment out the line that changes the coordinates of node 1020.
- Add to your script access_data.js a loop through all the nodes in the model, writing each node label to the dialog box using *Message*.
- Save and run the script.
- Now modify your loop, adding an “if” condition. We will try to fold the mesh along x=100:
  - First, stretch the mesh in the X direction by a factor of 2.
  - If a node’s x-coordinate (n.x) is 100, then the z-coordinate is increased by 5.
  - If a node’s x-coordinate (n.x) is >100, then the z-coordinate is increased by 10 and the x-coordinate becomes 100 – (n.x – 100)
- Save and run the script.
  Note that comparing floating point numbers sometimes needs a tolerance. e.g. `if (n.x > 99.9 && n.x < 100.1)` instead of `if (n.x == 100)`.
  In this case the nodes in “shell_mesh1.key” are exactly at 100 so == works.

In case of problems with this exercise, look at access_data_3_complete.js

Constructor

Node(Model*[Model]*, nid*[integer]*, x*[float]*, y*[float]*, z*[float]*, tc (optional)*[integer]*, rc (optional)*[integer]*)

Description

Create a new Node object.

Arguments

Name	Type	Description
Model	Model	Model that node will be created in
nid	integer	Node number
x	float	X coordinate
y	float	Y coordinate
z	float	Z coordinate
tc (optional)	integer	Translational constraint (0-7). If omitted tc will be set to 0.
rc (optional)	integer	Rotational constraint (0-7). If omitted rc will be set to 0.

Return type

Node object

Example

To create a new node in model m with label 100, at coordinates (20, 40, 10)

```
var n = new Node(m, 100, 20, 40, 10);
```

Within the description of each Class in the PRIMER JavaScript manual is a section telling you how to create new data of that type (the “Constructor”)

- For example, look at the script *make_shell_mesh.js*:

```
Message("Making nodes");

for (iy=0; iy<num_y+1; iy++)
{
    for (ix=0; ix<num_x+1; ix++)
    {
        x = x0 + ix*size_x;
        y = y0 + iy*size_y;
        z = 0.0;
        var n = new Node(m, ID_node_next, x, y, z);
        ID_node_next++;
    }
}
```

- Write a new script named *create_data.js* to loop through all the nodes in a model, adding a *LOAD_NODE Z-direction, loadcurve ID = 101, scale factor = 1.0.
- In this case, the loadcurve itself has not been defined. A “latent” definition will be created. The data for the curve may be added later.
- Check your script using the *model shell_mesh1.key*
- Make sure that the *LOAD_NODEs appear in PRIMER.
- If the loads do not appear, it is because the Entity switch for loads is off.
  - Add a statement to your script to turn it on – see Visibility in the Global class.

In case of problems with this exercise, look at create_data_complete.js

- These are part of Core JavaScript.
- For a full list of available functions, consult a JavaScript textbook
- Examples of maths and logic operations:

```
length = Math.sqrt(x*x + y*y + z*z);
test = Math.max(x, y);
x_abs = Math.abs(x);
```

- List of static functions belonging to the Math class:

abs, acos, asin, atan, atan2, ceil, cos, exp, floor, log, max, min, pow, random, round, sin, sqrt, tan

- Textbooks typically contain large sections on “Client Side”, which is irrelevant to PRIMER scripts. Only the Core sections are relevant.
- The textbook will be useful as a reference, for example to find what Math functions or (character) String functions are available and how to use them.
- Example: “JavaScript – The Definitive Guide” (5th edition) by David Flanagan, published by O’Reilly. ISBN 0-596-10199-6
- Online reference guides are also available – search for “Core JavaScript Reference”.

- In JavaScript, an array is actually an object with certain special properties.
- To declare a new array in JavaScript, use the Array Constructor:

```
var coords = new Array();
```

- Arrays can contain numbers, strings, or objects. It is not necessary to declare which type of data will be contained.
- It is not necessary to declare the size of the array – it will be automatically extended as needed. Optionally, you can declare the size:

```
var my_array = new Array(10);
```

- The array members start at index zero. Array members can be given values like this:

```
coords[0] = 1.5;
coords[1] = 2.0;
```

Note – square brackets are used to contain the array index

- …Or like this:

```
coords.push(1.5);
coords.push(2.0);
```

push assigns the next available array member

- To find the length of an array:

```
var l = coords.length;
```

- Different types of data may be stored in an array:

```
My_array[3] = 1.5;
My_array[4] = "coordinates";
My_array[5] = new Node(100,10,20,30);
```

- True multi-dimensional arrays are not supported by JavaScript, but you can set up an array of arrays. e.g.

```
My_array = new Array(10);
for (i=0; i<10; i++)  My_array[i] = new Array(10);
My_array[5][7] = 1.5;
```

- Files are treated as objects. Look at the File class in the PRIMER Javascript manual.

```
var f = new File(filename, File.READ);

var line =  f.ReadLine();  // read the first line

while ( line != undefined )
{
    (do something)
    line =  f.ReadLine();   // read the next line
}
f.Close();
```

Open a file

f.ReadLine reads the next line from the file

When f.ReadLine reads beyond the end of the file, the return value is undefined

Class constants

- File.READ is a constant belonging to the file class.
  - This tells the interpreter that we want to open the file for reading, not writing.
- Class constants are upper case.

```
var f = new File("nodes.csv", File.READ);
```

File class

The File class allows you to read and write text files. More...

Class functions

- Exists(filename[*string*])
- IsAbsolute(filename[*string*])
- IsDirectory(filename[*string*])
- IsFile(filename[*string*])
- Mkdir(directory[*string*])

Member functions

- Close()
- Flush()
- ReadChar()
- ReadLine()
- ReadLongLine()
- Write(string[*Any valid javascript type*])
- Writeln(string[*Any valid javascript type*])

File constants

Name	Description
File.APPEND	Flag to open file for appending
File.READ	Flag to open file for reading
File.WRITE	Flag to open file for writing

Exercise: Reading external files

- We will now write a script to read a comma-separated file.
- In a text editor, open the existing script *mesh_from_file_1.js.*
- Look at the File class in the PRIMER JavaScript manual.
- Where the comment line says “add code here”, add commands to open a file (using the variable *filename* and the file object *f*), and read the next line from the file
- Try to run the script. If it does not work, check in the dialog box for error messages.
- The most likely problem is that Primer looked in the wrong directory to find the file *nodes.csv*. To fix this, you could (but do not do it now) change the script such that the variable filename now includes the full path, e.g.

```
var filename = "C\\temp\\training\\nodes.csv";
```
Windows

```
var filename = "/data/training/nodes.csv";
```
Unix/Linux

- Note that back-slash is a special character, so a double-back-slash is needed (the first back-slash means “the next character is as-written, not a special character”).

In case of problems with this exercise, look at mesh_from_file_1_complete.js

- Rather than typing the full path and filename into your script, it is easier to browse for the file.
- In your script *mesh_from_file_1.js*, instead of setting "*var filename = …* ", use the function GetFilename (Window class) to ask the user to browse for the file. The function will return the filename as a variable.
- The filename returned by GetFilename will include the full path, so the file does not have to be in the same directory as the script.
- Save and run your script.

In case of problems with this exercise, look at mesh_from_file_1a_complete.js

- We have used this method for reading lines from a file:

```
line = f.ReadLine();
while ( line != undefined )
{
     (do something)
     line = f.ReadLine();
}
```

- An alternative to this coding would be:

```
while ( (line = f.ReadLine()) != undefined )
{
     (do something)
}
```


- This part of the statement sets *line* equal to the return-value of f.ReadLine.
- The assignment to *line* is done before evaluating whether to continue the While loop.

- Next we want to extract the node ID and coordinates from each line of data in the file.
- Here is an example line:

```
1016,50,10,0
```

Node ID, X, Y, Z

- Step 1: Break down the string into a series of small strings, each containing a number (we will use the comma delimiter).
  - The Core JavaScript function *split* can do this.
- Step 2: Read a number from each small string.
  - The Core JavaScript functions *parseInt* and *parseFloat* can do this.
- Most string-handling is done using Core JavaScript functions. However, we have added a small number of extra extension functions (e.g. *NumberToString*).

- Use *split* to divide *line* into an array of strings (we will call this array *words*):

```
var line;
var words = new Array();

while ( line != undefined )
{
   words = line.split(",");
   if (words.length == 4)
   {
       Message("label   = " + words[0]);
       Message("x-coord = " + words[1]);
       Message("y-coord = " + words[2]);
       Message("z-coord = " + words[3]);
   }
   line = f.ReadLine();
}
```

- The members of the array *words* are still character strings. We need to convert them into numbers.
- Use the core JavaScript functions *parseInt* and *parseFloat. e.g*:

```
label = parseInt(words[0]);
x     = parseFloat(words[1]);
```

- We also need to check whether the “word” that we have just tried to convert is a valid number. For example, it could have been from a comment line.
- If parseInt and parseFloat fail to find a number, they return a special value ‘NaN’ (Not a Number), for which we can test using the function isNaN().

```
if (isNan(x) )
{
        Error("Bad x coordinate " + words[1]);
        Exit();
}
```

Exercise – converting character strings

- We will continue with the example that reads data from a csv file. This data is intended to contain (Node ID, X, Y, Z).
- Start from the existing script *mesh_from_file_2.js*. This script will create a node from data stored in variables *label, x, y and z*.
- Lines of code need to be added where shown in the script ("… add code here")
- Add a command to split *line* into separate strings, storing these in the array *words* (use the function *split*)
- Add commands to read numerical values for *label, x, y and z* from the strings stored in *words* (use *parseFloat and parseInt*)
- Run the script. What happens? Modify the script to fix any problem.

In case of problems with this exercise, look at mesh_from_file_2_complete.js

```
//   Open file
var f = new File("nodes.key", File.WRITE);

f.Writeln("*NODE");

var n = Node.First(m);
while (n)
{
   f.Writeln(n.KeywordCards());
   n = n.Next();
}

f.Writeln("*END");
f.Close();
```

Open a new file

Write a text string to the file

Write LS-DYNA keyword format data (for a node)

Close the file

- There are many core JavaScript functions for processing strings. For more details, consult a JavaScript reference book.
- For example, if reading a line containing numbers in fixed fields, like an LS-DYNA keyword:

```
word[0] = line.slice(0,9);
```

Take characters 0 to 9 from *line* and copy them into *word[0].* Note that the first character is 0, not 1

- When writing data to a file, we often need to write numbers in a given format. Unlike languages such as Fortran and C, there are no Format statements in JavaScript. Instead, use functions such as number.toFixed, number.toExponential, etc.
- We have written a function to make this a little more convenient:

```
word[0] = NumberToString(n.nid, 8);
word[1] = NumberToString(n.x, 16);
f.Writeln(word[0] + word[1]);
```

Write node ID to 8-character string

Write node x to 16-character string

Write these to file

Creating data for non-supported keywords

- Some of LS-DYNA's keywords are not currently supported by Classes in the PRIMER Javascript interface.
- Non-supported keyword data may be created using this method:
  - Write a file containing the keyword data
  - Use the Import function (Model Class) to read it into the model.
- See example script *create_non_supported_kwd.js*

```
f = new File("./ibtmp.key", File.WRITE);
f.Writeln("*KEYWORD");
f.Writeln("*CONSTRAINED_EXTRA_NODES_NODE");
f.Writeln(NumberToString(part_id,10) + NumberToString(n1.nid,10));
f.Writeln("*END");
f.Close();
m.Import("ibtmp.key");
```

Open temporary file

Write keyword data to file

Close temporary file

Import data from file into model

- An alternative to the methods previously given for looping through all the entities in a model is to use GetAll:

```
var nodes = Node.GetAll(m);
for (i=0; i<nodes.length; i++)
{
   x-coord = nodes[i].x;
   (do something)
}
```

Nodes is an array of node objects

- If you will be looping through all the nodes several times this will be quicker. However, the memory requirement is larger – this may become significant for very large models.

- Example: if *s* is a shell element object, then *s.n1* is the LABEL of the first node – it is not a node object. Therefore, if you want to find the node's coordinates, do it like this:

```
var node_label_1 = s.n1;
var node_1 = Node.GetFromID(m,node_label_1);
```

node_label_1 is a number

node_1 is a node object

or

```
var node_1 = Node.GetFromID(m,s.n1);
```

then

```
var x_coord = node_1.x;
```

- This is always the case when one entity A refers to another entity B – the relevant property of A is always the label of B, not an object. This is so that you can still access and change data for entities for which we have not yet created classes, e.g. You can change EOSID on a PART but there is no EOS class yet.

- Be careful also when creating new keyword data:

```
var s = new Shell(m, pid, n1, n2, n3, n4);
```

n1, n2, n3, n4 must be node labels, NOT node objects

Using Flags

- A flag is a marker that can be set True or False for each entity (of any type) in the model.
- Flags are re-used by different functions in PRIMER, to save memory.
- The same flags are available in the JavaScript interface.
- Why do we need flags?
  - Example: Orient a part. We need to move each node belonging to the part.

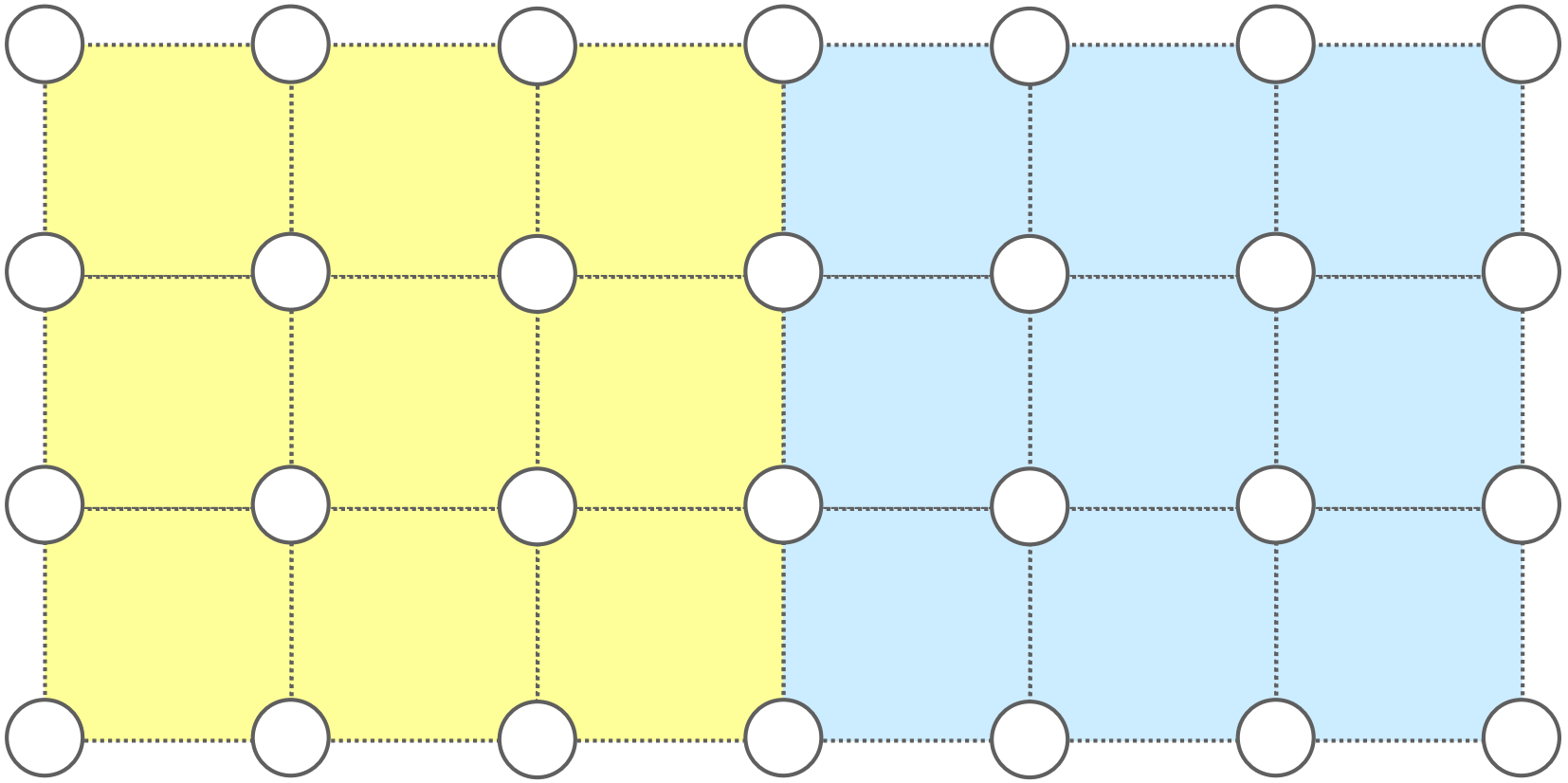
```
Loop through all elements belonging to the part
   Loop through all nodes on the element
      Move node
```

Wrong: nodes shared by >1 element will be moved more than once

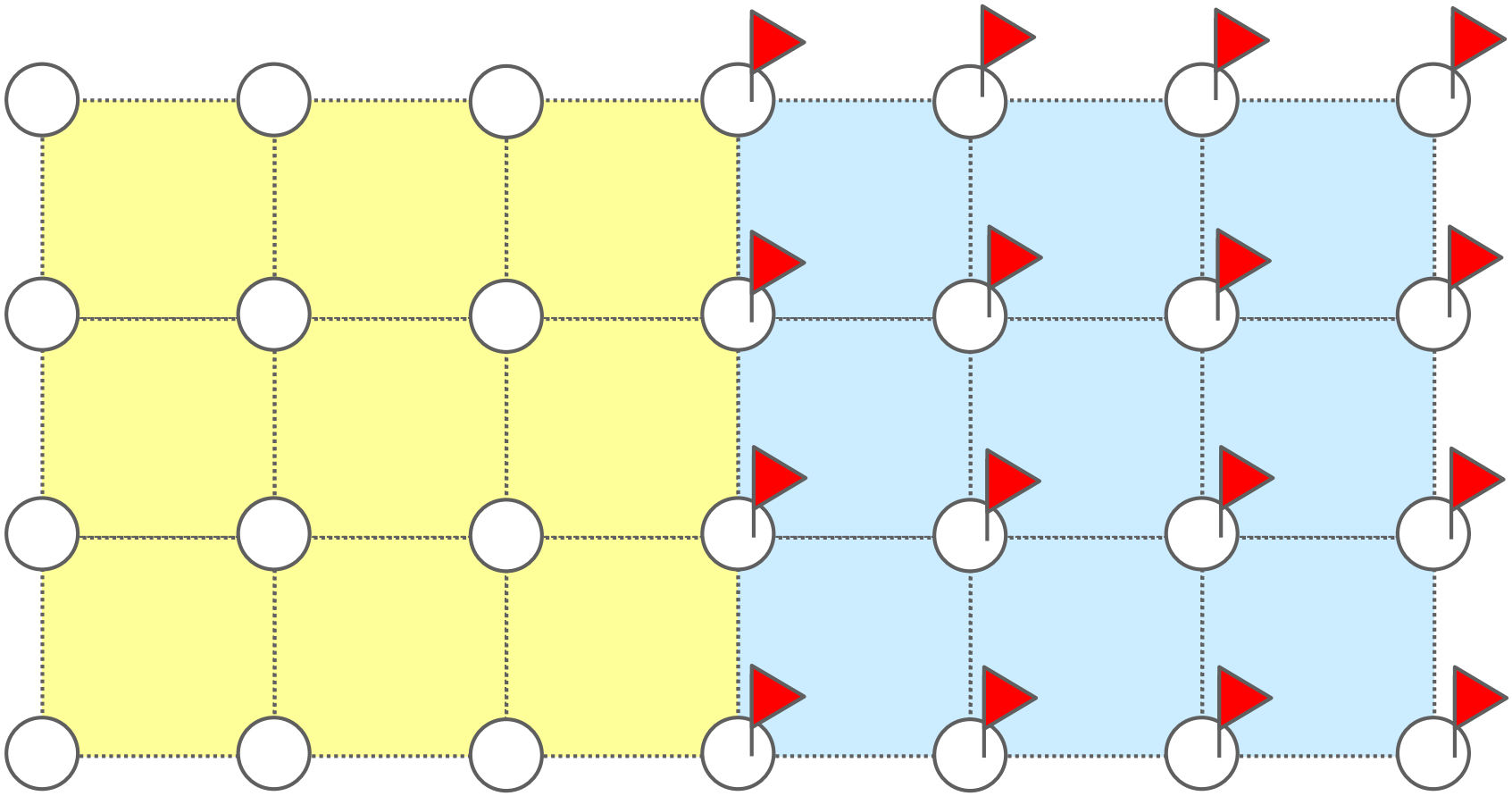
```
Loop through all elements belonging to the part
   Loop through all nodes on the element
      Flag the node
Loop through all nodes
   If node is flagged, move it.
```

Correct – each node belonging to the part is only moved once.

- Example – finding nodes on boundary between two parts meshed together

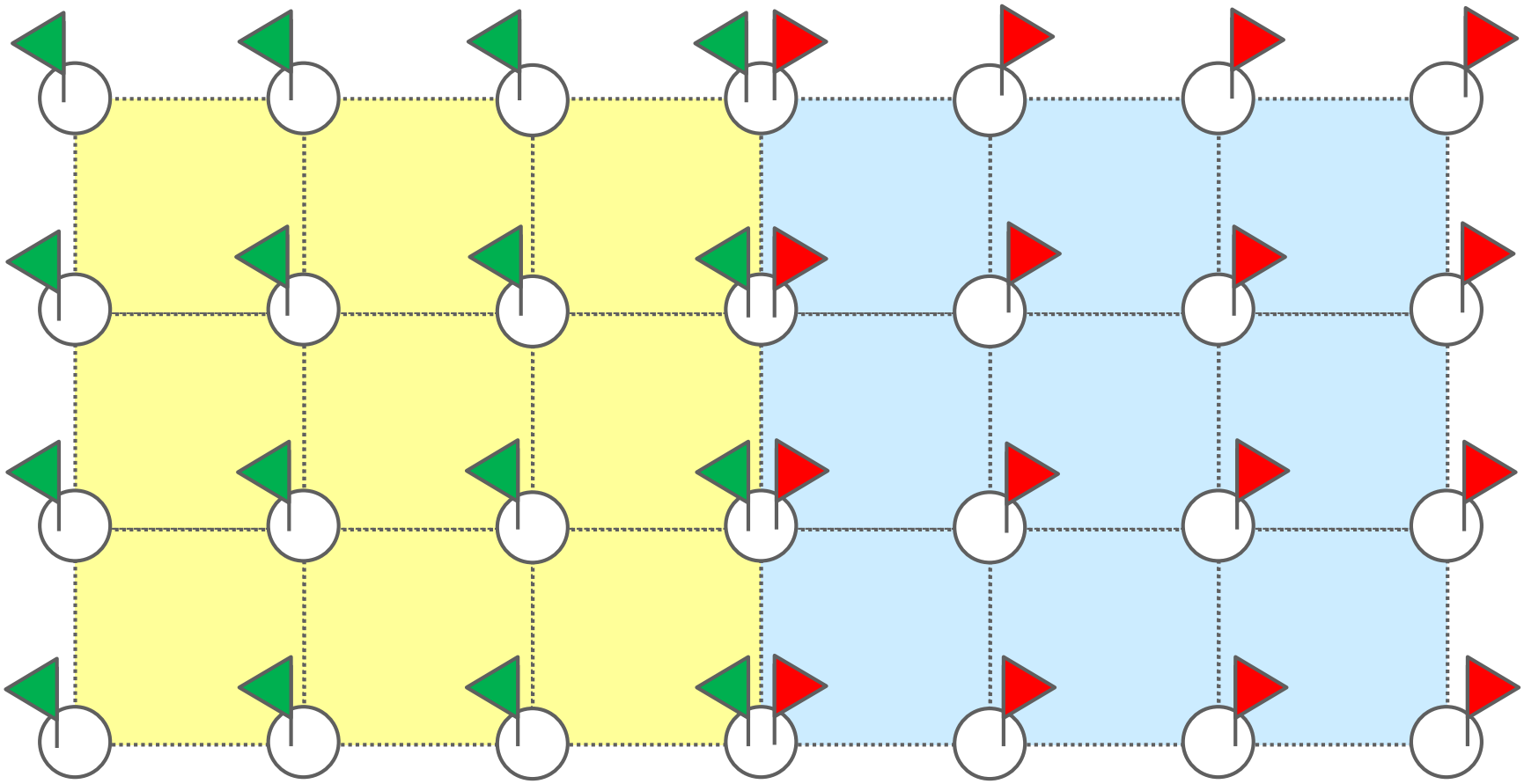


- Example – finding nodes on boundary between two parts meshed together



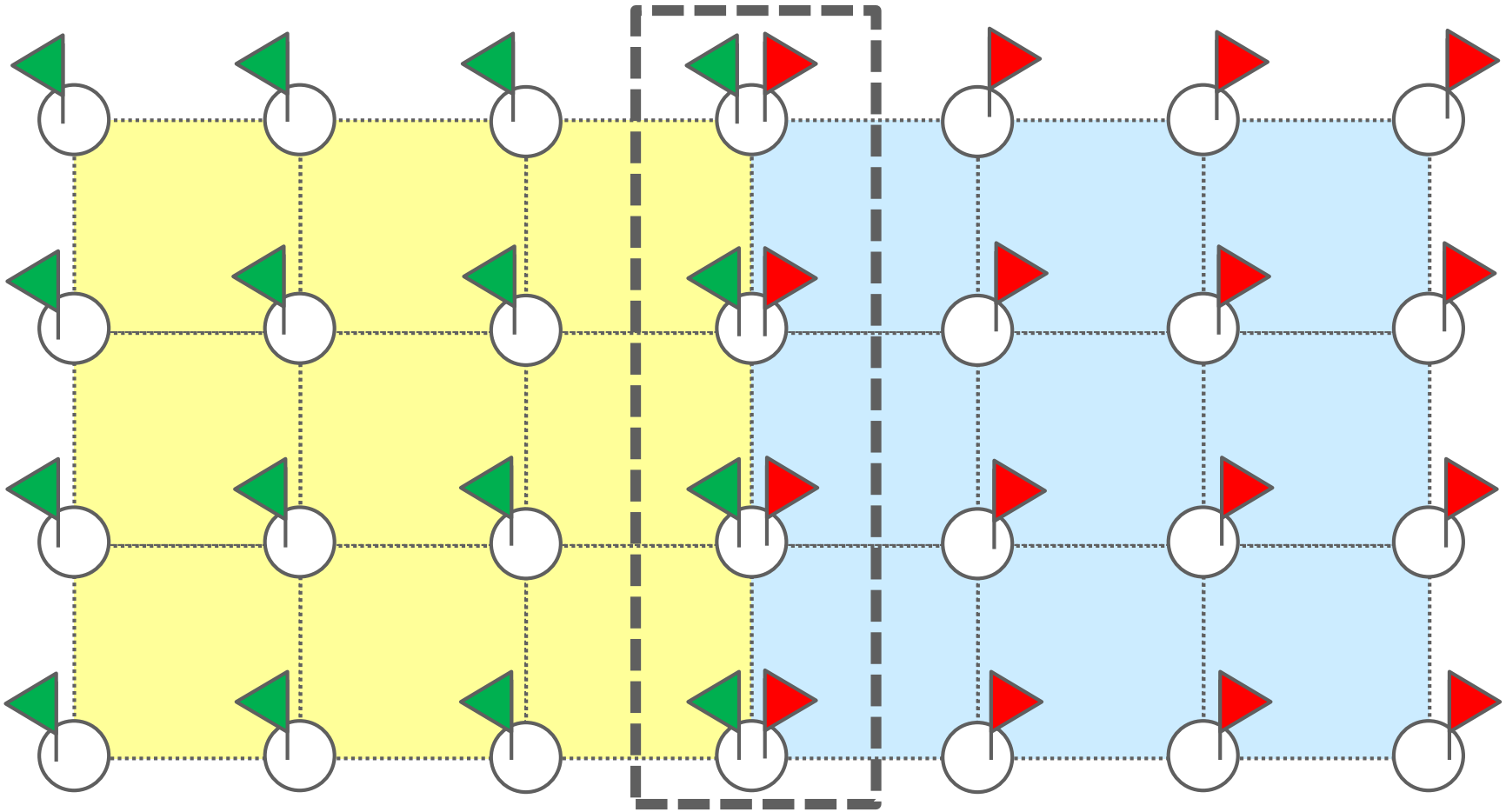
Flag nodes on one part with the red flag

- Example – finding nodes on boundary between two parts meshed together



Flag nodes on the other part with the green flag

- Example – finding nodes on boundary between two parts meshed together



Nodes flagged with both the red flag and the green flag are on the boundary

- There are many ready-made functions available that make use of flagging.
- More than one flag may be in use at any time. Since flags are used both by PRIMER itself and by the JavaScript interface, we must ask PRIMER to give us a flag.
- Important: ALWAYS Allocate and Clear a flag before using it, otherwise you could cause memory corruption in PRIMER.

```
var my_flag = AllocateFlag();
    m.ClearFlag(my_flag);
    my_part.SetFlag(my_flag);
```

Request PRIMER to give me a flag

Set the flag FALSE for all entities in Model m

Set the flag TRUE for the part object my_part

- Once you have finished with a flag you should return it:

```
ReturnFlag(my_flag);
```

- Note: AllocateFlag and ReturnFlag are in the Global class, while ClearFlag and some other flagging functions are in the Model class.

- The function PropagateFlag (in the Model class) finds all flagged entities, and “propagates” the flag to junior entities.
- For example, if a Part is flagged, then PropagateFlag will also set the flag for all elements of that part, and all nodes on those elements.

```
m.PropagateFlag(my_flag);
```

- If a Part Set is flagged, PropagateFlag sets the flag for all the Parts in the set, and then all the elements and nodes of those parts.
- To detect whether an entity is flagged, use the function Flagged, e.g. for a node look in the Node class for the Member function Flagged.

```
if (n.Flagged(my_flag))    (do something);
```

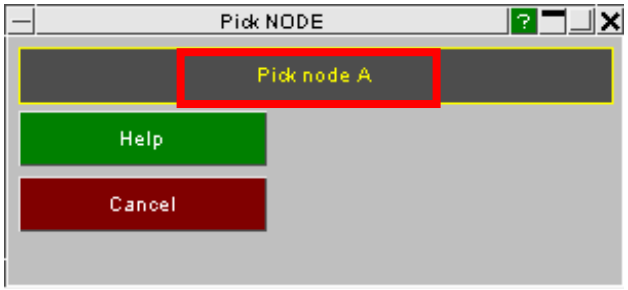
- To add flagged items to a set, use AddFlagged (in the Set class)

- Delete all models from PRIMER. Read in the model *front_of_car.key*.
- There is a partly-completed script *flagging.js*. Load this into a text editor.
- The objective is to identify parts with PID > 199999 and move them 1000mm in the x direction.
- Add code to do the following:
  - Allocate and clear a flag
  - Set the flag for parts whose Part ID is > 199999
  - Propagate the flag
  - Incrementing the x-coordinate of flagged nodes by 1000
  - Return the flag

In case of problems with this exercise, look at flagging_complete.js

- The function Pick is available for Parts, elements, nodes, etc. The function invokes PRIMER's picking capability - click on the entity in the graphics window.

```
n = Node.Pick("Pick Node A");
```



- The Pick function is for picking a single entity – to pick several entities, your script should call Pick each time an entity is to be picked:

```
na = Node.Pick("Pick Node A");

nb = Node.Pick("Pick Node B");
```

- If the user picks a node, the function returns a Node Object. If the user presses Cancel, it returns "undefined".

- Another example: Ask the user to pick multiple nodes, and store them in an array. We do not know how many nodes the user will pick.

```
var nodes = new Array();
var num = 0;
var n;
while (n = Node.Pick("Pick a node", m) )
{
   nodes[num] = n;
   num++
}
Message ("Number of nodes picked = " + num);
```

- When the user presses Cancel, the picking function will return null, so the *while* loop will finish.

- Start from *picking.js*.
- Delete all models from PRIMER, read in *front_of_car.key*.
- Add code to the script so that the user is asked to pick a part.
- The flag *my_flag* should be set true on that part.
- Check that the user did not press “Cancel” – if so, the part p would be null, and the script should exit.
- The picked parts will then be moved in X – this section of the script is already written.

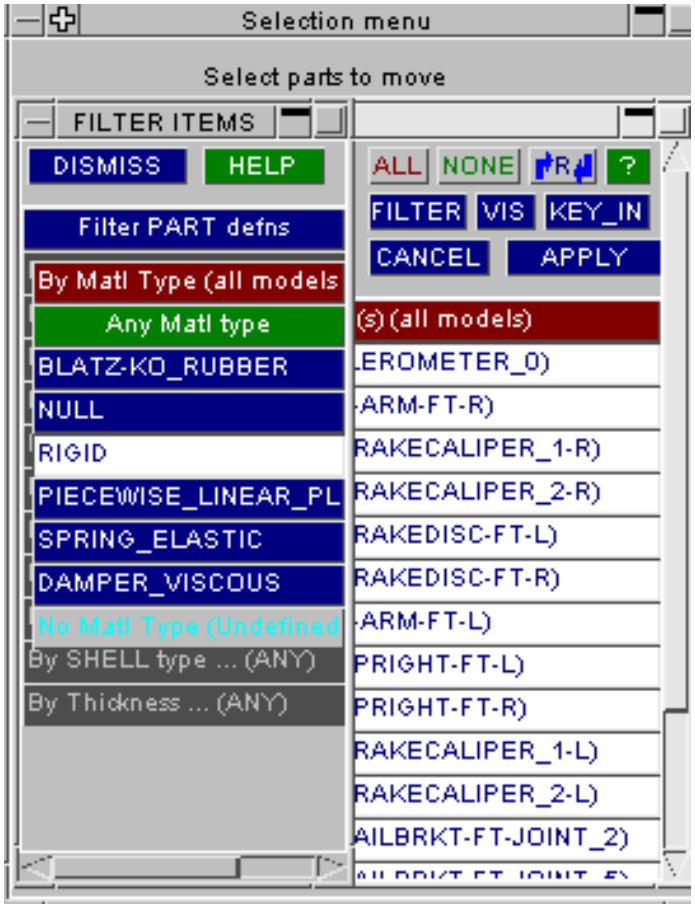
In case of problems with this exercise, look at picking_complete.js

- PRIMER's Object Menus can also be used for selecting entities. Object menu selection also allows VIS (and then pick or drag across a screen area), Filter, Key In, etc.
- The Select functions are very similar to Pick, except that, instead of returning one object of the type picked (e.g. Node.Pick returns one node), the Select function has no return value – instead, it sets a flag for all the entities selected. The flag is an input to the Select function:

```
var my_flag = AllocateFlag();
m.ClearFlag(my_flag);
Node.Select(my_flag,"Select Nodes to be processed");
var n = Node.First(m);
while (n)
{
   if (n.Flagged(my_flag)) (do something);
   n = n.Next();
}
```

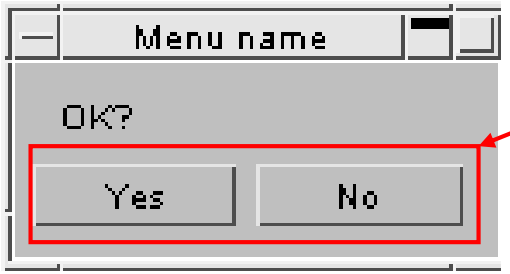
Exercise - Select

- Complete the script *select.js* – add a step to select parts.
- Test the script by running it, use the object menu to filter the parts by material type, choose MAT_RIGID, select all the parts, Apply.
- You should see in the dialog box that 15 parts were selected. The rigid parts should be moved, and put in a new set.



In case of problems with this exercise, look at select_complete.js

- Some ready-made windows are available: look in the PRIMER JavaScript manual in the Window Class. For example, Window.Message:



List of buttons to appear on menu, separated by | (bitwise “or”). The menu will disappear when one of these buttons is pressed. YES and NO are Class Constants. The available constants are shown in the manual.

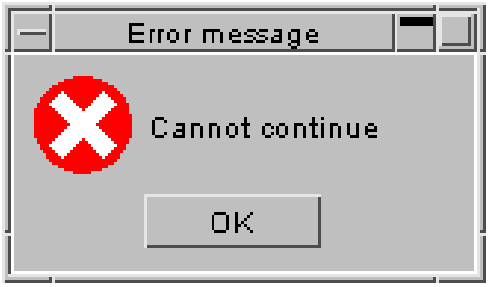
Return value is the Constant belonging to the button that was pressed

```
var Answer = Window.Message("Menu name","OK?", Window.YES | Window.NO);
if (Answer == Window.NO) Exit();
```

Name	Type	Description
title	string	Title for window.
question	string	Question to show in window.
buttons (optional)	constant	The buttons to use. Can be bitwise OR of Window.OK, Window.CANCEL, Window.YES or Window.NO. If this is omitted Yes and No button will be used. By default the window will be modal. If Window.NONMODAL is also given the window will be non-modal instead. Note that this will not currently allow windows created in javascript to be processed, it only applies to 'native' PRIMER menus.

- A simpler example uses only one button. In this case, the function's return value can be ignored (the script does not need to know which button was pressed:

```
Window.Error("Error message", "Cannot continue", Window.OK);
```



- By default, these windows prevent access to other functions in Primer until the user presses a button. However, this can be changed using the Constant NONMODAL.
- In this way the script can guide a user through an interactive process:

```
Window.Information("Instructions", "Step 1 - Model Check",
                                   Window.OK | Window.NONMODAL);
Window.Information("Instructions", "Step 2 - Create a Part",
                                   Window.OK | Window.NONMODAL);
Window.Information("Instructions", "Step 3 - Create a Part Set",
                                   Window.OK | Window.NONMODAL);
```

Exercise – using ready-made windows.

- Start from *ready_made_windows.js.*
- Add a check that model m exists. If not, use Window.Error to inform the user and then exit from the script. The window should contain an “OK” button only. Test this by deleting all models from Primer and running the script.
- After the user has picked a part, use Window.Question to check whether the user wants to move that part. In the script, the character-string forming the question is already done for you.
- The window should have “Yes” and “No” buttons. If the user presses “No”, exit.

In case of problems with this exercise, look at ready_made_windows_complete.js

- Most of the scripts in this training course assume that the model will be M1, e.g.

```
var m = Model.GetFromID(1);
```

- It is more convenient to use *Model.Select*:

```
var m = Model.Select("Select a model");
```

- If there is only one model present in PRIMER, that model will be selected without asking the user any questions. This will still work if the model is M2 or M3, etc.
- If there are no models in PRIMER, an error message will be issued.
- If more than one model is present, the user will be asked to select a model.
- Please use this function in your script and test it:
  - a) when no models are present
  - b) when one model is present
  - c) when two or more models are present.

- Any PRIMER capability that can be accessed via command line (commands typed in the dialog box, or written in a command file) can be issued from a JavaScript, using the functions DialogueInput or DialogueInputNoEcho.
- For example:

```
DialogueInput("/CHECK checkfile check.dat apply");
```

- After each call to DialogueInput (or DialogueInputNoEcho), PRIMER returns automatically to the main menu. If you need to issue a sequence of commands without returning to the main menu, use a single call to with multiple arguments (separated by commas):

```
DialogueInputNoEcho("/mech point " + pt_name,
        "position " + points[i].x + " * " + points[i].z,
        "done", "accept")
```

Exercise: Command-line commands

- In Primer's dialog box, type *H* (that means *Help*), to see the available command-line commands.
- We will use the command PART_INFO to write the part table data to a csv file.
- Make sure that you have a Model 1 in Primer.
- Try typing this into Primer's dialog box:

```
/PART_INFO WRITE data.csv 1
```

- Check the csv file using Excel.
- In your script *ready_made_windows.js*, add a call to DialogueInput to use the above command.
- It is better to use *m.number* rather than assuming that Model 1 will be present:

```
DialogueInput("/PART_INFO WRITE data.csv " + m.number);
```

- JavaScript is case sensitive. *Node* is not the same as *node*.
- When testing to see if something is equal you must use ==. i.e.

```
if (i == 10)      // This tests if i is 10
if (i = 10)       // This sets i to 10.
                  // The test is then if(10) which is always true
```

- Do not forget semi-colon after each line
- Remember that the first member of an array is array[0], not array[1]
- Always declare variables with var or you may overwrite variables by mistake.
- If you read data from a file then you are reading strings. If you want numbers you need to use parseInt() and parseFloat() to read the number from the string.
- When calling Extension functions, check carefully whether the input arguments and return values should be entity labels or objects.
- Do not mix up Member functions and Class functions:

```
n = Node.Next();          //  Wrong – Next is a Member function! Needs a
node
                          //  object, not the class name
n = n.Next();             //  Correct
```

- When calling Class functions, don't omit the class

```
PropagateFlag(my_flag);        //  wrong
m.PropagateFlag(my_flag);      //  correct
```

- Include lines of code to check the return values from functions. Many functions return null if something cannot be found or done.

```
na = Node.Pick("Pick Node A");
Message ("Node ID = " + na.nid);
```

Wrong - if the user presses Cancel on the picking menu, *na* will be null. This will cause an error.

```
na = Node.Pick("Pick Node A");
if (na) Message ("Node ID = " + na.nid);
```

Correct - *na* is tested for null value before being used

- Do not create an infinite loop with *while*:

```
n = Node.First(m);
while (n)
{
   Message ("Node ID = " + n.nid);
}
```

Wrong – the line *n = n.Next();* has been forgotten. Each time through the loop, n is still the first node in the model! If you run this script, you will have to crash Primer to escape.

- And finally - always write lots of comments!

- Try to run the script *script_with_errors.js*
- When the script asks you to select a file, find *nodes.csv* (as used in the read-from-file example.
- Find and correct the errors. After correcting an error, try to run the script again.
- In case of trouble with this exercise, check the next slides or compare with *script_with_errors_corrected.js*.

```
var filename = Window.Getfilename("Select file", "CSV file?", "csv");
```

wrong spelling of function name – F should be upper-case

```
// Create node
        var n = new Node(model,label,x,y,z);
```

variable name was m, not model

```
// Get the node label from the first word
        var label = words[0];
```

words[0] is a string. Need parseInt(words[0])

```
//  Close the file
f.Close()
```

missing semi-colon

```
var n1 = node.First(m);
var node1 = n.nid;
Message ("Node label and x-coordinate: " + node1 + ", " + node1.x);

//   Loop through nodes, printing labels and incrementing x-coord
while(n1)
{
   Message("Node label = " + n1.nid);
   n1.x = n1.x + 100.0;
}

while(n1)
{
   Message("Node label = " + n1.nid);
   n1.y = n1.y + 100.0;
   n1 = Node.Next();
}
```

mis-spelling of Node class name

variable name was n1, not n

need the node object, not its label

infinite loop – missing n1 = n1.Next();

Next is a Member function, not a Class function

```
//   Get all nodes into an array
var nodes = Node.GetAll(m);

//   Create a shell element using the 1st, 2nd, 12th and 11th node
var s = new Shell(1001,100,nodes[1],nodes[2],nodes[12],nodes[11]);

Visibility("Node", true);
UpdateGraphics();
```

First node is nodes[0], not [1]

Model object is needed here

Need node IDs not node objects (e.g. nodes[0].nid)

UpdateGraphics is a Member function in the Model class – needs *m.UpdateGraphics();*

PRIMER JavaScript – Part 3

- Look at the Set class in the PRIMER JavaScript manual.
- To create a set and add some entities to it (this can be done only for SET_..._LIST, not _GENERATE, _ADD, etc).

```
var s = new Set(m, 100, Set.PART);
```

s is a Set object

```
s.Add(part_id_1);

s.Add(part_id_2);
```

- To loop through the entities in an existing set (this works with all set types, including _GENERATE, _ADD, etc)

```
var s = Set.GetFromID(m, 100, Set.PART);
```

Find the *SET_PART with ID 100

```
s.StartSpool();
while (part_id = s.Spool() )
```

Loop through all the parts in the set

```
{
   do something
}
```

- Functions are blocks of code that may be called from your main program or from other functions – in some languages, these would be called subroutines.
- Functions may be added to your script, after the main body of the script.
- Functions are often used to avoid repeating the same code several times.

```
…
var z_axis = new Array();
z_axis = CrossProd(x_axis,y_axis);
y_axis = CrossProd(z_axis,x_axis);
…

function CrossProd(a,b)
{
    var c = new Array();
    c[0] = a[1]*b[2] - a[2]*b[1];
    c[1] = a[2]*b[0] - a[0]*b[2];
    c[2] = a[0]*b[1] - a[1]*b[0];
    return c;
}
```

Main body of the script

No semi-colon

Braces

Function named CrossProd is within the same text file, after the end of the main body of the script. It can be called from the main body of the script, or from within another function.

- Variables, arrays, and objects may be input to a function via the list of arguments:

```
z_axis = CrossProd(x_axis, y_axis);
y_axis = CrossProd(z_axis, x_axis);

function CrossProd(a, b)
{
   var c = new Array(3);
   c[0] = a[1]*b[2] - a[2]*b[1];
   …
   return c;
}
```

- An input argument does not have to have the same name within the function as in the main body of the script.
- The types of the arguments do not have to be declared within the function – the interpreter knows these automatically from the type of the arguments where the function is called.

- Alternatively, input may be variables (constants, arrays, or objects) that were declared in the main body of the script – these are “global”, i.e. they can be “seen” by all functions within the same text file.

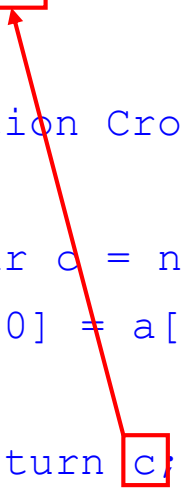
```
var x_axis = new Array();
var y_axis = new Array();
…
var z_axis = CrossProdXY();

function CrossProdXY()
{
   var c = new Array(3);
   c[0] = x_axis[1]*y_axis[2] - x_axis[2]*y_axis[1];
   …
   return c;
}
```

Because x_axis and y_axis are declared in the main body of the script, they are automatically visible to all functions within the same text file.

Brackets are still needed even when there are no arguments

- The return statement allows a single variable (e.g. a constant, array, or object) to be passed back to the calling statement:

```
z_axis = CrossProd(x_axis, y_axis);
y_axis = CrossProd(z_axis, x_axis);

function CrossProd(a, b)
{
   var c = new Array(3);
   c[0] = a[1]*b[2] - a[2]*b[1];
   …
   return c;
}
```



- The type of the return value does not have to be declared – the interpreter knows what type of variable is being returned by the function. For example, in this case z_axis and y_axis are arrays because the returned variable, c, is an array.

- If an array appears in the argument list, and its members are changed by the function, the new values are passed back to the calling statement, e.g. in the example below, `z_axis` and then `y_axis` will take the values calculated for the array `c` in the function:

```
CrossProd(x_axis, y_axis, z_axis);
CrossProd(z_axis, x_axis, y_axis);

function CrossProd(a, b, c)
{
      c[0] = a[1]*b[2] - a[2]*b[1];
   …
}
```

- Objects can be passed in the same way, and their properties assigned within the function. For example, a node object could be passed into a function via the argument list, and its coordinates could be changed within the function.

- This does not work if the input variables are constants, e.g.

```
var x = 1.0;
AddOne(x);
Message("x = " + x);

function AddOne(value)
{
   value = value + 1.0;
   Message ("value = " + value);
}
```

The result of this will be "x = 1.0" – the change to x that occurred within the function AddOne is not passed back to the calling statement.

- This behaviour is similar to C and some other languages – it occurs because what is passed to the function is the value of the variable, rather than the variable itself.

- Global variables, even if they are constants, may be changed within a function; the changed values are visible in the main body and all other functions. For example:

```
var x = 1.0;
AddOneToX();
Message("x = " + x);

function AddOneToX()
{
   x = x + 1.0;
   Message ("x = " + x);
}
```

The result of this will be "x = 2.0" – the change to global variable x that occurred within the function AddOneToX is visible everywhere.

- The *scope* of a variable is the region of the program in which it is defined. A *global* variable has global scope; it is defined everywhere in your JavaScript code. Variables declared in a function are defined only within the body of the function. They are *local* variables and have local scope.
- In a function a local variable takes precedence over a global variable with the same name. If you declare a local variable with the same name as a global variable you hide the global variable. e.g.

```
var scope = "global";

function checkscope()
{
        var scope = "local";
        Message(scope);
}

checkscope();           // prints "local"
Message(scope);         // prints "global"
```

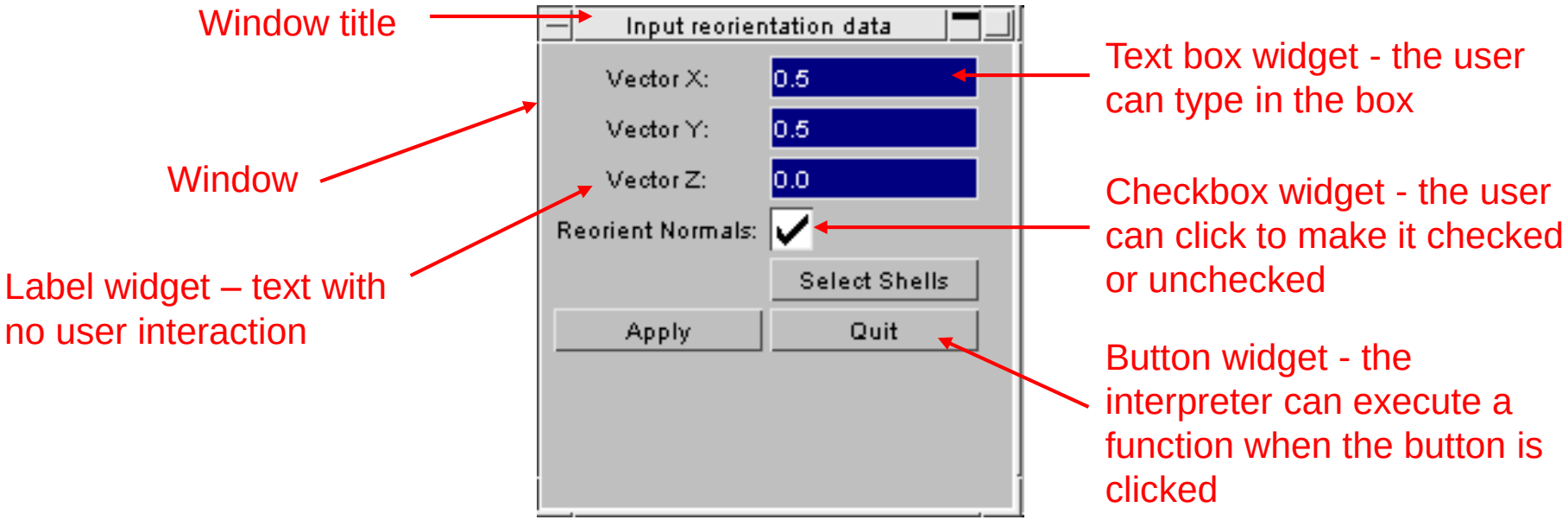
- Always use var to declare variables. If you don't use it for local variables you will overwrite the global variable e.g.

```
scope = "global";              // variable declared without var is
global

function checkscope()
{
    scope = "local";           // we have changed global variable, not
local one
    Message(scope);
}

checkscope();                  // prints "local"
Message(scope);                // prints "local"
```

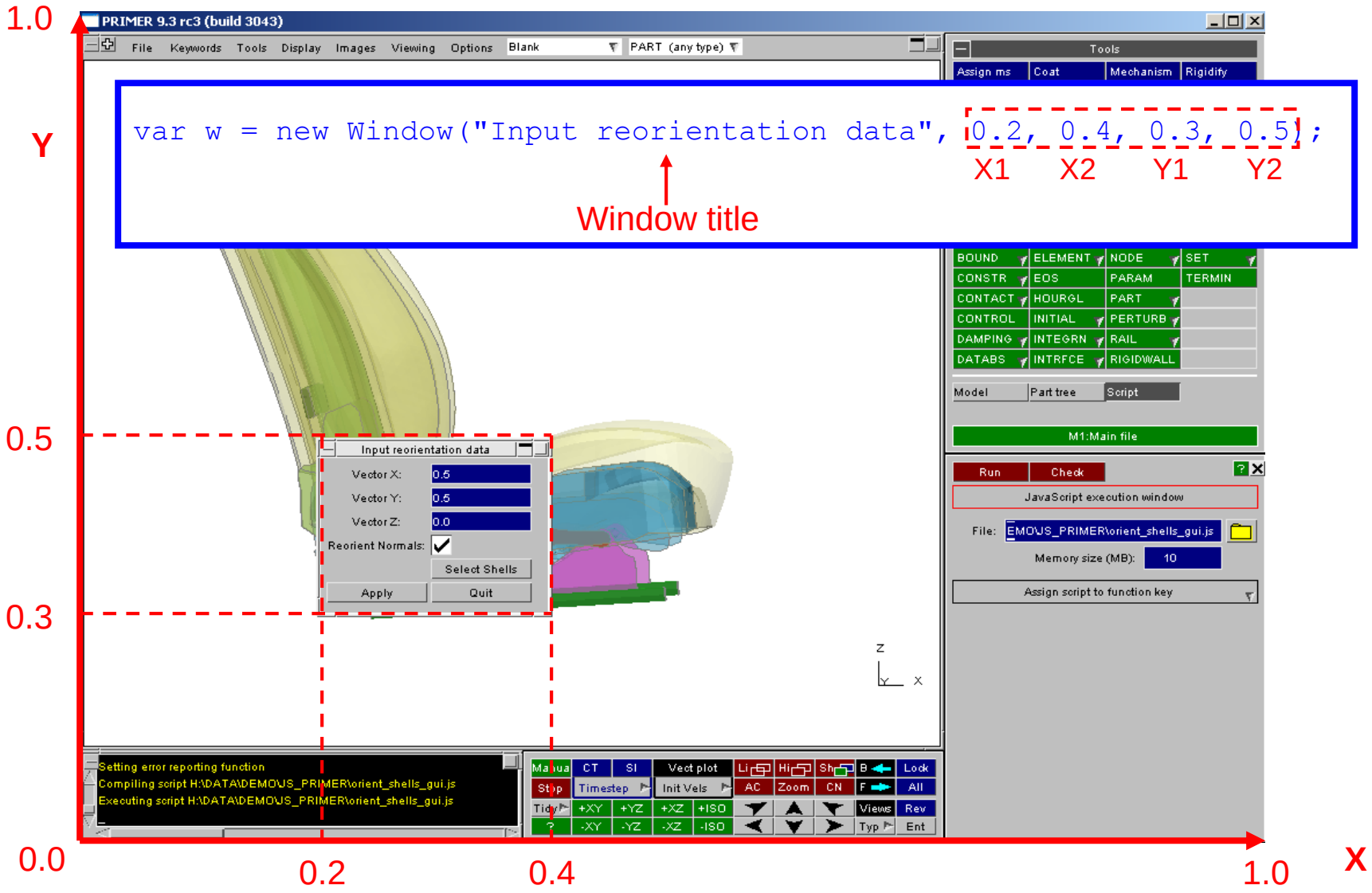
- Starting from *function.js*, write a new function to count the selected parts. Most of the code for this is already included at the bottom of the file.
- Call your new function where indicated in the main body of the script.
- Run the script. Check that the correct number of selected parts was echoed to the dialog box.

In case of problems with this exercise, look at function_complete.js

- New menus may be created using the Window and Widget classes.
  - “Window” = the window containing a floating menu
  - “Widget” = buttons, text boxes, text labels, checkboxes, etc
- See PRIMER Javascript manual for details.
- The functions and techniques are identical in D3PLOT and PRIMER



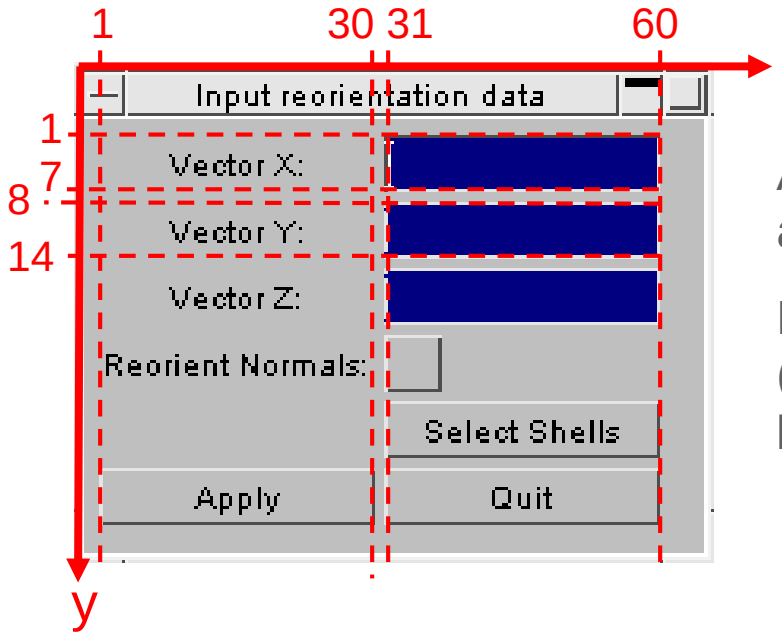
Creating a window



Adding Label and Textbox Widgets to a Window

```
var w = new Window("Input reorientation data", 0.2, 0.4, 0.3, 0.5);
var label_X = new Widget(w, Widget.LABEL,     1, 30,  1,  7, "Vector X:");
var text_X  = new Widget(w, Widget.TEXTBOX,  31, 60,  1,  7, "");
var label_Y = new Widget(w, Widget.LABEL,     1, 30,  8, 14, "Vector Y:");
var text_Y  = new Widget(w, Widget.TEXTBOX,  31, 60,  8, 14, "");
                                             X1  X2   Y1  Y2
```

Initial value of text



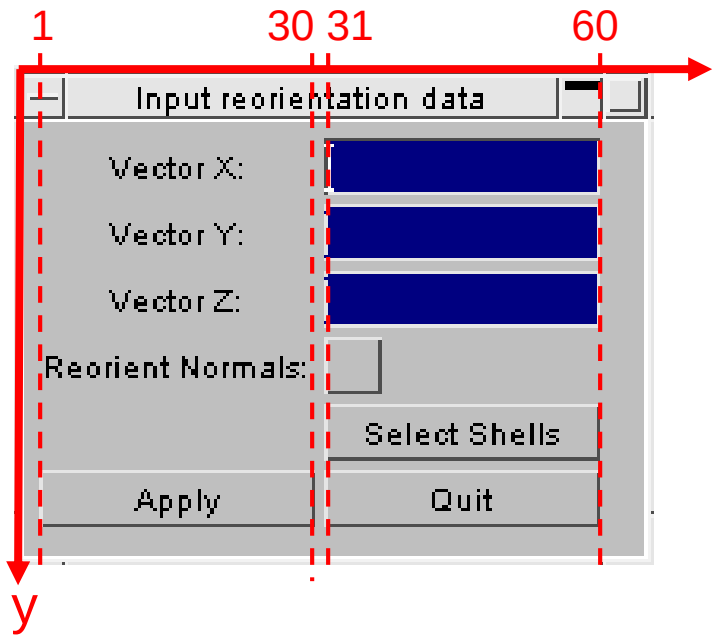
All widgets have a type (e.g. Widget.TEXTBOX) and (X,Y) coordinates within the window.

Note that each widget is a variable with a name (e.g. text_X) – the variable names will be used later to identify the user's input.

Dimensions are dependent on 'Display Factor' in Menu Attributes.

```
var text_R  = new Widget(w, Widget.LABEL, 1, 30, 22, 28, "Reorient Normals:");
var chkbx_R = new Widget(w, Widget.CHECKBOX,  31, 37, 22, 28);
var s_button = new Widget(w, Widget.BUTTON, 31, 60, 29, 35, "Select Shells");
var a_button = new Widget(w, Widget.BUTTON,   1, 30, 36, 42, "Apply");
var q_button  = new Widget(w, Widget.BUTTON,  31, 60, 36, 42, "Quit");
                                              X1  X2  Y1  Y2
```

Text on button



- To display the menu and wait for user input, use the function Show (Window class):

```
w.Show();
```

- To make something happen when the user presses a button, set the property onClick for each button. This should reference the name of a function without arguments. The functions referenced can be either existing JavaScript functions (e.g. Exit) or functions included within your script.

```
var s_button = new Widget(w, Widget.BUTTON, 31, 60, 29, 35, "Select Shells");

s_button.onClick = MySelectFunction;

var q_button  = new Widget(w, Widget.BUTTON,  31, 60, 36, 42, "Quit");

q_button.onClick = Exit;
```

- To change the colour of a button, use the property *background*. The colour of the text can be changed using the property *foreground*:

```
a_button.background = Widget.DARKRED;

a_button.foreground = Widget.WHITE;
```

- As explained above, the function called when a button is pressed cannot have any arguments.
- If you want to call a function that does have arguments, use a “wrapper” function as in this example:

Variable my_flag is global (declared in main body of the script), so is “visible” to all functions below

```
var my_flag = AllocateFlag();
apply_button = new Widget(w, Widget.BUTTON, 31, 60, 29, 35, "Apply");
apply_button.onClick = MyApplyFunction;

function MyApplyFunction()
{
   MyOtherFunction(my_flag);
}
```

MyApplyFunction has no arguments, and so can be called by the Apply button. It calls MyOtherFunction using global variables as arguments.

- To read the contents of a text box, use the property *text*, which is a character variable (string). To convert to a number, use *parseInt* or *parseFloat*:

```
var input_X  = new Widget(w, Widget.TEXTBOX,  31, 60,  1,  7, "");
var x = parseFloat(input_X.text);
```

- Note that the user is expected to change the contents of the text box. The user might do this several times. We need to use the final value only. One way to achieve this is to read from the text box in the function that is activated by the Apply button:

```
var a_button = new Widget(w, Widget.BUTTON, 31, 60, 29, 35, "Apply");
a_button.onClick = MyApplyFunction;
function MyApplyFunction()
{
   var x = parseFloat(input_X.text);
   (do something with x)
}
```

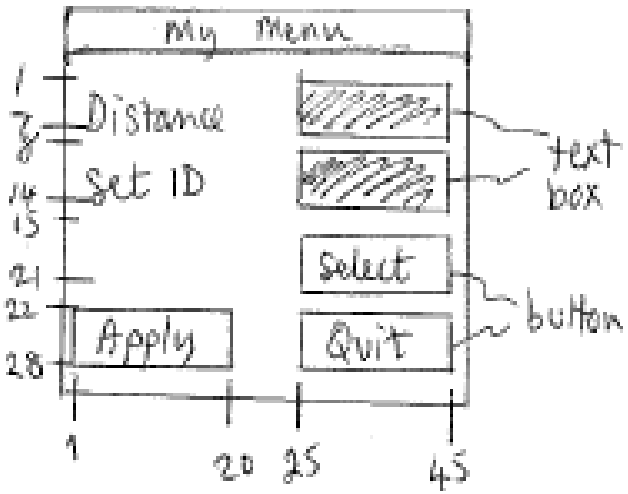
- It is often useful to write a function that can be called whenever the user changes any input (text box, checkbox, etc). Purposes of the function may include:
  - Read numerical values from text boxes
  - Count and display how many items the user has selected
  - Make buttons active or inactive according to what inputs have been supplied

```
input_X.onChange = MenuUpdate;
input_Y.onChange = MenuUpdate;
function MenuUpdate()
{
   if (input_X.text != "" && input_Y.text != "")
   {
      x = parseFloat(input_X.text);
      y = parseFloat(input_Y.text);
      apply_button.active = true;
   }
}
```

- If the same function is used for the ‘onClick’ or ‘onChange’ property on several Widgets how can you tell which Widget the user has clicked or changed? The answer is to use a special JavaScript keyword called ‘this’ which is set to the Widget. e.g. If we wanted to change the background colour of the `input_X` widget to be red if the user changes it.

```
input_X.onChange = MenuUpdate;
input_Y.onChange = MenuUpdate;

function MenuUpdate()
{
   if (this == input_X) input_X.background = Widget.DARKRED;

   if (input_X.text != "" && input_Y.text != "")
   {
      x = parseFloat(input_X.text);
      y = parseFloat(input_Y.text);
      apply_button.active = true;
   }
}
```

- We will add a GUI to an existing script.
- The function of the script is similar to the previous example on Selection: selected parts will be moved by a distance in X, and put in a new Part Set.
- User inputs will be:
  - Translation distance for the selected parts
  - ID of part set to be created
  - Button “Select” for the user to Select Parts
  - Button “Apply” to perform the moving and set creation
  - Button “Quit” to leave the menu without performing any action
- The first step is to sketch the menu and annotate the position of the Widgets



Example sketch – decide your own menu layout

- Start from *script_GUI.js*
- Add code where indicated to create a window with title “My Menu”.
- Try to make the menu appear in the bottom-right corner of the screen.
- At this stage, do not add any other code to the script.
- What happens when you run it?
- Add *w.Show();* to display the window.
- Run the script. What happens?
  - If there are no Widgets on the menu, there is no way for the user to supply input. The w.Show function waits indefinitely for input. You will have to crash out of PRIMER.
- Add a Quit or Cancel button. When this button is clicked, it should call the function Exit (use the onClick property of the Quit button).
- Run the script and check that the menu appears, and disappears when you press Quit.
- Next, add the other widgets and check that they appear in the correct places.

- For the Select button, write a “wrapper” function that calls *Part.Select* using the existing flag *my_flag*
- When the Apply button is pressed, call the existing function *MoveParts*
- Add code to function *MoveParts* to read the input values of Distance (variable x_trans) and Set ID (variable setID_start) from the text boxes.
- Add code to check that neither of the text boxes is blank; if so, return to the main body of the script and wait for further user input.
- The function already counts how many parts were selected (variable *num_selected*). Add code so that if no parts were selected, issue an error message (function Error) and return to the main menu an wait for further user input.

In case of problems with this exercise, look at script_GUI_complete.js

- If time allows…
- Add colour to the buttons (suggestion: make the Apply button DARKRED with WHITE text)
- Make the Apply button inactive (greyed out) initially (use the property *active*). Whenever a text box is changed, and after the Select function has been used, call a new function MenuUpdate that checks the input. If OK, then make the Apply button active.
  - Checks are: the text in the text boxes is not blank; at least one part has been selected.

In case of problems with this exercise, look at script_GUI_2_complete.js

Further topics for Self-Study

- When variables are created memory is used in the computer to store the data. Strings, arrays and objects do not have fixed size so storage must be allocated for them when one is created.
- In some programming languages this memory must be freed manually. In JavaScript this is done automatically for you using a technique called *garbage collection*. For example:

```
var s = "Hello, world!";                    // memory allocated for string
    var n = new Node(m, 100, 20, 40, 10);   // memory allocated for Node object
    s = n;                                  // Overwrite s with n
```

- After this has run the string "Hello, world!" is no longer reachable (there are no references to it). The interpreter detects this and frees up the space to be used again.
- This means that you can create a script that makes lots of temporary objects, arrays or strings without worrying about memory.

- New objects may be created, and properties added to them, as shown below.
- The type of contents or list of properties does not have to be defined in advance.

```
var my_object = new Object();

my_object.title = "Checking information";

my_object.ID = 3;

my_object.node = new Node(m,1000,10.0,20.0,30.0);

Message("X-ccord of my node = " + my_object.node.x);
```

- It is also possible to add new properties to existing objects, such as node or element objects. These properties have no effect on the model data in PRIMER, they exist only temporarily while the script is being run.

```
var s = Shell.GetFromID(m,1001);

s.Checked = "no";
```

- The Xrefs class is used to find the references from one entity to others.
  - To find nodes connected to a shell, the information is already there (s.n1 etc).
  - To find the shells connected to a node, we need to use Xrefs.
- In this example, we start with a Node object (my_node) and set the flag my_flag for all the shells attached to my_node.

```
var xrefs = my_node.Xrefs();

num = xrefs.GetTotal("SHELL");

for (count=0; count<num; count++)
{
   var id = xrefs.GetID("SHELL", count);
   var shl = Shell.GetFromID(model, id);
   shl.SetFlag(my_flag);
}
```

Get the Xrefs object for this node

How many Xrefs are of type SHELL?

Get the ID of each shell

Get the shell object and set the flag

- XML is often used for data files as it is flexible and can be extended as necessary. e.g. perhaps there is a connection file with the format:

```
<connections>
  <connection type="spotweld">
    <position>10, 20, 30</position>
  </connection>
 <connection type="rivet">
    <position>20, 20, 30</position>
  </connection>
</connections>
```

- This is a basic XML file that contains elements, attributes and text.
- PRIMER implements a simple stream-oriented parser where you declare functions to call when things are found in the XML file. e.g.
  - The start of an element
  - The end of an element
  - Text

- For example this will print the contents of a basic XML file

```
// Create a new parser object
var p = new XMLParser();

// assign handlers
p.startElementHandler  = startElem;
p.endElementHandler    = endElem;
p.characterDataHandler = text;

// parse the file
p.Parse("/data/test.xml");

function startElem(name, attr)
{
// handler to be called when the start of an element is found
    Println("START: " + name);

    // Print attributes
    for (n in attr)
    {
        Println(" attr: " + n + "=" + attr[n]);
    }
}
```

```
function endElem(name)
{
   // handler to be called when the end element of an element is
   // found
    Println("END: " + name);
}

function text(str)
{
   // handler to be called when text is found
    Println("TEXT: '" + str + "'");
}
```

- If a handler is not defined then that part of the XML document is skipped.
- For example if the `characterDataHandler` was not defined the text in the XML file would be skipped.
- For more details see the XMLParser class or contact Oasys

- JavaScript has regular expressions to allow you to do pattern matching on strings. e.g.

```
var s = "node 10   x=1.2   y=20   z=0";
var regex = /(\d+)\s+x=(.+)\s+y=(.+)\s+z=(.+)/
var result = s.match(regex);
if (result != null)
{
    var fullmatch = result[0];      // contains "10  x=1.2  y=20
z=0"
    var id        = result[1];      // contains "10"
    var x         = result[2];      // contains "1.2"
    var y         = result[3];      // contains "20"
    var z         = result[4];      // contains "0"
}
```

- '\d' means "match a digit", '\s' means "match whitespace" (space or tab), '.' means "any character", '+' means "one or more of the previous thing" and () means "save what you have matched".
- For more details see a good JavaScript textbook.

- Numbers and booleans (true/false) are “primitive types” in JavaScript.
  A primitive type has a fixed size in memory. For example a number occupies 8 bytes of memory in JavaScript.
  If each variable in JavaScript reserves 8 bytes of memory, the variable can directly store the primitive value.
- Objects, arrays and functions are reference types.
  Objects, for example, can be any length. They do not have a fixed size.
  Since they do not have a fixed size, the data cannot be stored directly in the 8 bytes of memory for the variable.
  Instead the variable stores a *reference* to the data (a pointer or an address where the data can be found)
- Strings are a special case. They do not have a fixed size but they can be treated as a primitive type (see a good JavaScript book for more details)
- The reason this is important is that primitive types and reference types behave differently when they are copied or passed to functions.
  Primitive types are manipulated by *value*.
  Reference types are manipulated by *reference*.

- Primitive types are copied by value

```
var m = 1;      // variable m holds the value 1
var n = m;      // Copy by value. n holds a distinct value 1
```

- Primitive types are passed by value. This could cause functions to work differently to what you expect.

```
var total = 0;
var m = 1;
function add_value(total, num)
{
        total = total + num;
}
add_value(total, m);
Println(total) // prints 0, not 1
```

- This is because when passing by value, the *value* is copied, so the variable *total* above used in the function is a copy and you only change the internal value of the copy, not the original variable.

- Reference types are copied by reference

```
var a = new Array(1,2,3);  // a is an Array so is a reference type
var b = a;                 // Copy reference to new variable
a[0] = 99;                 // Change original array
Println(b)                 // prints 99,2,3
```

- This may not be what you expected!
- Remember that *a* contains a reference to where the array data is stored. After the second line there is still only one array object, but *b* now also contains the reference to the same array.
- If you want to make a distinct copy of an array you have to do something like

```
var a = new Array(1,2,3);
var b = new Array();
for (var i=0; i<a.length; i++) b.push(a[i]);
```

- Reference types are passed by reference.

```
var totals = new Array(1,2,3);
var num = 10;

function add_to_totals(totals, num)
{
    totals[0] += num;
}

add_to_totals(totals, num);
Println(totals) // prints 11,2,3
```

- In the above example a reference to *totals* is passed to the function add_to_totals, so updating totals in the function updates the same array.

Training course

JavaScript for Oasys PRIMER and D3PLOT