Maplets

A Customizable Interface to Maple

Stephen Forrest, Maplesoft

sforrest@maplesoft.com

First Steps

Introduction to Maplets

What is a Maplet?

A Maplet consists of one or more windows which interact with the user by means of buttons, checkboxes, text fields, and other standard graphical controls.

Why bother with Maplets?

Maple is a powerful problem-solving environment capable of doing quite advanced mathematics, but the price of this power is considerable complexity.

With a Maplet, this complexity can be retained but restrained: a simple menu interface, tailored to a specific task or problem.

Examples

With these built-in Maple commands which launch Maplets, we will see a bit of what Maplets are capable of.

Computing Matrix Inverses

The following Maplet is designed for students learning to compute matrix inverses using Gaussian elimination; the point-and-click interface avoids much of the confusion over syntax beginning users face.

```
> Student:-LinearAlgebra:-InverseTutor():
```

```
>
```

This is not to say, however, that Maplets are designed to be used only by beginners. Building an interface customized for a specific task offers convenience and benefit even to those users who would otherwise have had no difficulty in performing the task in Maple.

Solving Differential Equations

In the following example we illustrate the use of a Maplet in solving a differential equation.

```
> eqn := diff(y(x),x,x)-10*(1-y(x)^2)*diff(y(x),x)+y(x) = 0;
```

```
> dsolve[interactive]( { eqn, y(0) = 2, D(y)(0) = 0 } ) ;
```

```
>
```

Building Plots

In this example we show how a Maplet may be used for plotting a function.

```
> plots[interactive]( x^2-y^2 );
```

```
>
```

First Maplet

```
> restart;
```

In this first example, we see a Maplet with one window. This window contains the text My First Maplet in the title bar, and in homage to the grand old tradition of programming tutorials, the body of the window has the text Hello, World!.

```
> use Maplets:-Elements in
      maplet := Maplet( Window( "My First Maplet", [
          ["Hello, world!"]
      ] ) );
  end use:
```

```
> Maplets:-Display( maplet );
```

```
>
```

First, notice the use Maplets:-Elements in part. The Maple identifiers used in building a Maplet are members of a Maple package named Maplets:-Elements , which is itself a subpackage of the package Maplets . We could have written the above code using full names (e.g. Maplets:-Elements:-Window) but for convenience we've enclosed the code in a use block. We'll see this throughout the examples.

Notice also that there are two separate Maple commands here: building a Maplet with a call to the Maplet command and assigning it to the name maplet , and actually running it with the Maplets:-Display command.

A Simple Layout

The visual design of the body of the window is created using a list of lists. Each list forms a separate row in the resulting window body. It is up to the Maplet design engine to place these objects in what is hopefully a visually pleasing arrangement.

```
> use Maplets:-Elements in
      maplet := Maplet( Window( "My Second Maplet", [
          ["Hello There"],
          ["1", "2", "3"],
          ["ABCDEFG", "HIJKLMNOPQR"],
          [Button( "OK", Shutdown() )]
      ] ) );
  end use:
```

```
  Maplets:-Display( maplet );
>
```

It is possible to exert finer control over the placement of objects in a window, but for most Maplets, the default is more than sufficient. We'll see later how we can make this layout more interesting.

The first non-text object in the above maplet is the button. That button has the text OK and when that button is clicked, the Shutdown action occurs.

There are many other such visual elements which can be placed into a Maplet window.

More Interesting Elements

In the following Maplet we see a number of the more interesting elements available for use.

```
> use Maplets:-Elements in
      maplet := Maplet( Window( "My 3rd Maplet", [
          [Button( "OK", Shutdown() ),
           CheckBox( "Some text" ),
           ComboBox( "Combo", ["Hello", "Goodbye"] ),
           DropDownBox( "DropDown",
              ["DropDown", "Hello", "Goodbye"]
           )
          ],
          ["ABCD", Label( "ABCD" )],
          [ListBox( ["First", "Second", "Third", "4th"] )],
          [MathMLViewer( sin(x)^2 + cos(x)^2 )],
          [Plotter( plot( sin(x), x=0..10 ) )],
          [Slider( 10..20, 15, showticks = true, minorticks = 1 ),
           TextBox(),
           TextField(),
           ToggleButton("Press Here")]
      ] ) );
  end use:
  Maplets:-Display( maplet );
>
```

Element Options

We've seen that we can choose from a wide variety of visual elements to include in a Maplet window. It's natural to ask how we can customize these elements to suit our particular needs.

Each Maplet element has a number of options which affect its behaviour and appearance. In looking at the previous Maplet, you might have noticed that one element was specified with an argument `showticks = true`. Other options modifying Maplet elements can be specified in similar ways.

```
> ?Button
```

```
> use Maplets:-Elements in
      maplet := Maplet(
          Window( "A Colourful Button",
```

```
            [[Button( "OK", background = blue, enabled = false,
                tooltip = "Can't click me", Shutdown() )]]
        )
    );
end use:
Maplets:-Display( maplet );
```

```
>
```

A note on style: in general, one should be reasonable about customizing options. As much as you might love to colour an important button red, you probably shouldn't: users won't expect it, and they'll be uncomfortable if they see it.

References and State

Now, we can do much more than simply customizing the display of option. A number of Maplet elements are capable of holding state, such as the TextField in the following Maplet, which a user can modify:

```
> use Maplets:-Elements in
    maplet := Maplet( Window( "Text Maplet", [
        [TextField()],
        [Button( "OK", Shutdown() )]
    ] ) );
end use:
Maplets:-Display( maplet );
```

```
>
```

A more complete list is:

CheckBox
ComboBox
ListBox
MathMLEditor
Slider
TextBox
TextField
ToggleButton

The state information associated with a Maplet element is not restricted to things which a user can directory modify. For example, an element 'knows' whether or not it is visible.

We might like to programmatically ask the Maplet about one of its elements. In order to refer to such elements in order to ask such questions, we must identify them. We do this using references. In the previous example, we could replace the code `TextField()` swith `TextField["input"]`, which allows this element to be referenced later by the name "input".

The reference may be a Maple string or symbol. Every element can be given a reference by indexing the element with the desired name or string. In general, it is safer to work with strings, but like everything else in Maple, symbols too are allowed.

The first use of a reference will be in the Shutdown action (of which we will see more later on):

```
> use Maplets:-Elements in
      maplet := Maplet( Window( "Text Maplet", [
          [TextField["input"]()],
          [Button( "OK", Shutdown( ["input"] ) )]
      ] ) );
  end use:
  Maplets:-Display( maplet );
>
```

Here, when the user clicks on OK, then the Shutdown action is triggered and returns a list. The one object in this returned list is the contents, or value, of the text field box referenced by **input**.

```
> use Maplets:-Elements in
      maplet := Maplet( Window( "Text Maplet", [
          [TextField[input]()],
          [TextField[input2]()],
          [Button( "OK",
             Shutdown( [input, input2, input(visible)] ) )
          ]
      ] ) );
  end use:
  Maplets:-Display( maplet );
>
```

What is 'visible'? Most elements have a default value option but each maplet has many other options. See TextField for a list of the options accepted by TextField.

From Maple code, you can access any of these options by using the convention **some_reference(option_name)**. If you do not include option_name with parentheses after specifying **some_reference**, and the element has a option named **value**, then that option is used.

Exercise 1

Part 1

Write a maplet which queries the user for his or her name, then returns it to the worksheet.

Part 2

Take this maplet and wrap it in a procedure which takes the returned name and prints Your name starts with X, where X is the first letter in the name which was returned.

Actions and Commands (Event Driven Programming)

With Maplets, you can define a default set of events (called actions) which occur when a maplet starts up. After that, nothing happens unless the user initiates an action by either clicking a button, moving a slider, or entering text in a field.

Most of the names of these action elements are rather self explanatory:

CloseWindow
Evaluate
RunDialog
RunWindow
SetOption
Shutdown

Shutdown exits a maplet, while **CloseWindow** only closes a maplet. By default, when the last window of a maplet is closed, the maplet exits and control is returned to the worksheet.

```
> use Maplets:-Elements in
      maplet := Maplet( Window( "My 4th Maplet", [
          ["Hello World"],
          [Button( "OK", Shutdown() )]
      ] ) );
  end use:
  Maplets:-Display( maplet );
```

```
>
```

We've seen before that we can give the Shutdown element a list of references to return. In addition to a list, it accepts a string as well; this is returned to the worksheet.

```
> use Maplets:-Elements in
      maplet := Maplet( Window( "My 5th Maplet", [
          ["Hello World"],
          [Button( "OK 1", Shutdown("Button 1") ),
           Button( "OK 2", Shutdown("Button 2") )]
      ] ) );
  end use:
  Maplets:-Display( maplet );
```

```
>
```

If you are returning both a value and a list of references, you will get an expression sequence with the first element being the return value and the second being a list of the references.

The next, and most important, command element is Evaluate. This calls a Maple procedure and allows the maplet to interact with the Maple kernel and library.

```
> Sure := proc()
     print( "Sure was pressed" );
  end proc:

  Maybe := proc()
     print( "Maybe was pressed" );
  end proc:

  use Maplets:-Elements in
      maplet := Maplet( Window( "My 5th Maplet", [
          [Button( "Sure", Evaluate( function = 'Sure()' ) ),
           Button( "Maybe", Evaluate( function = 'Maybe()' ) )
```

```
        ],
        [Button( "Exit", Shutdown() )]
    ] ) );
  end use:
  Maplets:-Display( maplet );
>
```

The SetOption element can be used to directly set an option of any maplet element within the maplet:

```
> use Maplets:-Elements in
    maplet := Maplet( Window( "Set Option", [
        [Button( "Enable",  SetOption( exit(enabled) = true ) ),
         Button( "Disable", SetOption( exit(enabled) = false ) ),
         Button[exit]( "Exit", Shutdown() )]
    ] ) );
  end use:
  Maplets:-Display( maplet );
>
```

You can use SetOption to clear a text field quite easily:

```
> use Maplets:-Elements in
    maplet := Maplet( Window( "Set Option", [
        [TextField[TF](),
         Button( "Clear", SetOption( TF = "" ) ),
         Button[exit]( "Exit", Shutdown() )]
    ] ) );
  end use:
  Maplets:-Display( maplet );
>
```

Communication between Maplets and the Maple Kernel

Once you call a Maple function, you can both query and update anything in a maplet. You do this by using the two tools Maplets:-Tools:-Get and Maplets:-Tools:-Set .

```
> use Maplets:-Elements in
    maplet := Maplet( Window( "Integrator", [
        ["Integrand: ", TextField[integrand]()],
        ["Integral:  ", TextField[integral]( editable = false )],
        [Button( "Integrate",
             Evaluate( function = 'Integrate()' )
         ),
         Button( "OK", Shutdown([integral]) )]
    ] ) );
  end use:
```

```
> Integrate := proc()
    local expr;
```

```
        expr := Maplets:-Tools:-Get( integrand::algebraic );
        expr := int( expr, x );
        Maplets:-Tools:-Set( integral::algebraic = expr );
    end proc:
    Maplets:-Display( maplet );
>
```

Error Handling

Let's take a look at that last example. What happens if the user does something wrong?

The most important tool we have for this purpose is the try-catch construct:

```
> Integrate := proc()
        local expr;
        try
            expr := Maplets:-Tools:-Get( integrand::algebraic );
        catch:
            return; # message?
        end try;

        try
            expr := int( expr, x );
        catch:
            return; # message?
        end try;

        Maplets:-Tools:-Set( integral::algebraic = expr );
    end proc:
> Maplets:-Display( maplet );
>
```

Exercise 2

Part 1

Ask the user to enter an integer n in a TextBox. Let the user test whether n is prime by clicking on a button, and update another TextBox with this information.

Part 2

To the previous Maplet, add a DropDownBox with which the user can choose his or her favourite prime factor of the number n. Make this dropdown box inactive when the Maplet starts, and activate it and fill its contents when the user tests the primality of n.

Extra: Activate the dropdown box only if n has more than one prime factor.

The Finer Details

More than One Window

Up until now, we've seen only one window. A Maplet is not just a window: a window is merely one component of a Maplet, and so therefore it is reasonable that a Maplet has more than one window.

Now, if you only include one window, the Maplet assumes you want to run that window on starting. If you have more than one, it's not sure, and returns an error message:

```
> use Maplets:-Elements in
      maplet := Maplet(
         Window( "Hi",  [["Hi"]] ),
         Window( "Bye", [["Bye"]] )
      );
  end use:
```

This introduces us to more complex actions. In the following Maplet, we specify which window we'd like to start with:

```
> use Maplets:-Elements in
      maplet := Maplet( onstartup = RunWindow( W1 ),
        Window[W1]( "Hi", [["Hi"]] ),
        Window[W2]( "Bye", [["Bye"]] )
      );
  end use:
  Maplets:-Display( maplet );
```

```
>
```

Below is a slightly more complicated example. Here we have two commands running as a result of one action: clicking the button.

```
> use Maplets:-Elements in
      maplet := Maplet( onstartup = RunWindow( W1 ),
          Window[W1]( "Hi", [[
              Button( "OK",
                  Action( CloseWindow( W1 ), RunWindow( W2 ))
              )
          ]] ),
          Window[W2]( "Bye", [[Button( "Okay", Shutdown() )]] )
      );
  end use:
  Maplets:-Display( maplet );
```

```
>
```

Dialogs

Maplets come with a number of preset dialog elements.

```
> ?Maplets,DialogElements
```

```
> use Maplets:-Elements in
      maplet := Maplet( AlertDialog(
```

```
          "Assuming x > 0 leads to a contradiction",
          'onapprove' = Shutdown("true"),
          'oncancel' = Shutdown("FAIL")
      ) );
  end use:
  Maplets[Display](maplet);
>
```

One thing to note: dialog elements do not have a state, so you cannot query the value that it returns.

```
> use Maplets:-Elements in
      maplet := Maplet( ColorDialog[D1](
          'onapprove' = Shutdown( [D1] ),
          'oncancel' = Shutdown()
      ) );
  end use:
  Maplets[Display]( maplet );
>
```

Menus

Menus are just organized buttons. It is recommended that you follow standard conventions if you decide to include menus in your maplet.

```
> use Maplets:-Elements in
      maplet := Maplet( onstartup = RunWindow( W1 ),
           Window[W1]( "With Menu",
           MenuBar( Menu( "File",
               MenuItem( "Hello", RunDialog( D1 ) ),
               MenuItem( "Exit", Shutdown() )
           ) ),
           [["Some Text"]]
      ), ConfirmDialog[D1]( "Are you fine?" ) );
  end use:
  Maplets:-Display( maplet );
>
```

You can use the menu separator to place a line between adjacent menu items. Use these to group common menu items and don't over do it...

Toolbars

Toolbars are like menus, except that they have buttons, as you can see in the worksheet environment.

```
> use Maplets:-Elements in
     maplet := Maplet(
        Window('title' = "Integration w.r.t. x",
           'toolbar' = ToolBar(
               ToolBarButton("Exit", Shutdown()),
```

```
                ToolBarSeparator(),
                ToolBarButton("Do It",
                    'onclick'=Evaluate('TF1' = 'int(TF1, x)'))
            ),
            [TextField['TF1'](),
             Button("OK", Shutdown("OK"))]
        )
    ):
end use:
Maplets[Display](maplet):
```

```
>
```

Pop-up Menus

The text field and text box elements takes a popupmenu option, which specifies a PopupMenu element.

```
> use Maplets:-Elements in
    maplet := Maplet( Window( "Integrator", [
        ["Integrand: ", TextField[integrand](
            popupmenu = PopupMenu(
                MenuItem( "Integrate",
                    Evaluate( function = 'Integrate()' ) )
            )
        )],
        ["Integral:  ", TextField[integral]( editable = false )],
        [Button( "OK", Shutdown([integral]) )]
    ] ) );
end use:
```

```
> Integrate := proc()
    local expr;
    expr := Maplets:-Tools:-Get( integrand::algebraic );
    expr := int( expr, x );
    Maplets:-Tools:-Set( integral::algebraic = expr );
end proc:
```

```
> Maplets:-Display( maplet );
```

```
>
```

Fancier Layouts

Okay, what I said before about formatting windows wasn't the whole truth.

We don't have to have a list of lists. Instead, what Maple does is interpret the first list as a list of elements to be displayed vertically. If that list contains any sublists, those are to be displayed horizontally, and so on, always alternating between vertical and horizontal display:

```
> use Maplets:-Elements in
    maplet := Maplet( Window( "Fancy", [
        "ABC",
```

```
            ["DEF", "GHI", ["J", "K", "L"]],
            ["M", ["OP", "QR", ["S", "T", "U"]]]
        ] ) );
    end use:
```

```
> Maplets:-Display( maplet );
```

Borders and Captions

We can add borders and captions to the various rows and columns of maplet elements.

```
> use Maplets:-Elements in
      maplet := Maplet( Window( "Fancy", [
          "ABC",
          ["DEF", "GHI", BoxRow( border = true, "J", "K", "L" )],
          ["M", BoxRow( border = true, caption = "Hello",
                        "OP", "QR", ["S", "T", "U"] )]
      ] ) );
    end use:
```

```
> Maplets:-Display( maplet );
```

```
> ?Maplets,LayoutElements
```

```
> plots[interactive]( sin(x) );
```

Elements in Detail

```
> ?Maplets,WindowElements
```

```
>
```

More Exercises

The following are suggested exercises to help you learn about Maplets. Choose any of the following exercises that interest you, or write your own Maplet that does something else entirely.

Exercise 3

Write a procedure which, takes two subsets of {a, b, c, d, e} and asks the user if the first is a subset of the second. Allow the user to click Yes or No, and open a new Maplet which indicates whether the user was right or wrong.

If the user was wrong, optionally try to discuss why he or she was wrong. E.g., {a, b, c} is not a subset of {a, b, d} because c is not an element of {a, b, d}.

Exercise 4

Pick a Maple function and write a Maplet interface to that function. See Maplets[Examples] for some

ideas on design of your Maplet.

Exercise 5

As the user to integrate a random polynomial (you can generate one by using the randpoly command).

Check the user's answer using whatever Maple tool you find appropriate. Tell the user if he or she is correct.

Exercise 6

Play the high-low game with the user. Select a random number from 0 to 100, and ask the user to guess the value. Tell the user how far off he or she was from the optimal number of guesses.

Exercise 7

Use the slider zoom in on a plot.