Eclipse™ Programming Supplement

The purpose of this document is to familiarize you with the power of Eclipse's programming features. Included are sample tools and tips for creating your own tools and constants.

Tutorial

Building a tool or defining a constant for use in Eclipse is quite easy. After completing this simple tutorial you should be ready to create your own tools and constants for tailoring Eclipse to fit your individual needs. To best use this tutorial it is best to have your Newton MessagePad nearby with Eclipse up and running. For a description of the features of Eclipse and the Eclipse Code Toolbox, refer to the manual.

Getting Started

First familiarize yourself with the buttons and lists in the Code Toolbox. There are many built-in features that will speed up the programming process.

We are going to build a simple function that calculates the area of a cone. It will accept two parameters, radius and height.

Defining ConicArea

To define a tool we must first enter the Code Toolbox by tapping the Code button. We assign a name to the tool by entering it in the field labeled name. Write *ConicArea* in the Name field. You can also use the keyboard or other handwriting recognition utilities to enter the name. Once the name is defined, make sure the Tool(Function) option is selected. If it isn't, tap Tool. Now we are ready to write the code for *ConicArea*.

Coding ConicArea

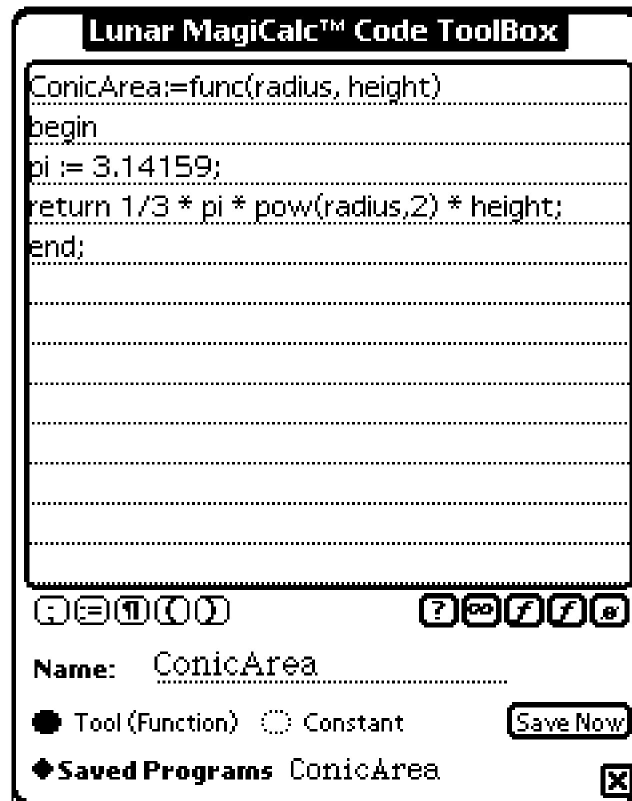
To begin coding, tap the Setup button. The toolbox window should now contain the shell for our new function, with *ConicArea* at the top. First we need to specify the parameters for the function. Double tap between the parenthesis after the word *func*.

Enter *radius*, *height* into the parenthesis. Now we have told Eclipse that *ConicArea* will accept two parameters, radius and height.

Between the *begin* and *end* statements is where the actual code goes to define the tool. On the line below *begin* enter the following:

```
pi := 3.14159;
return 1/3 * pi * pow(radius,2) * height;
```

The Code Toolbox should look like the figure on the right.



View of new tool, ConicArea after code definition.

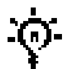
To save the tool you can either tap Save Now or the close box. If all went well Eclipse should display Tool Defined back in the Calculator display.

Using ConicArea

Now that the tool is defined, it will appear in the User pick list. To try out your new tool, select it from the pick list and tap in the values for the radius and height parameters separated by commas. Then tap solve.

Example:
:ConicArea(7,12) <Solve>
615.75164

ConicArea can now be used just as any other function in Eclipse would be. *ConicArea* is a simple example of how easy it is to program Eclipse. Browse the following section to see some more examples of possible functions.


Note: To erase a tool or constant, scrub out everything in the Code Toolbox for that function and tap Save Now or the close box.

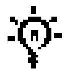
Example Routines

Calculating the Area of a Sphere

```
sphere_area:=func(r)         // where r is the radius of the sphere
begin
  return 4/3*3.14159*pow(r,3);    // 4/3 * pi * r^3
end;
```

example in calling this routine:

```
3*:sphere_area(5)+60
```



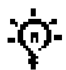
Note: // denotes a comment in the code. Any text after // will not be compiled by Eclipse. It is for reference when looking at the code.

Calculating the Parallel Resistance of an Unknown Number of Resistors

```
parallel_res:=func(resarray)  // where resarray is an array of values
begin
  numerator:=1;             // initialize variables
  denominator:=0;
  for i:=0 to length(resarray)-1 do    // length returns the value from 1 not 0
  begin
    numerator:=numerator*resarray[i];
    denominator:=denominator+resarray[i];
  end;
  return (numerator/denominator);
end;
```

same procedure, better way:

```
parallel_res:=func(resarray)
begin
  numerator:=0;
  denominator:=1;
  foreach element in resarray do  // assigns element to each point in resarray
  begin
    numerator:=numerator*element;
    denominator:=denominator+element;
  end;
  return (numerator/denominator);
end;
```



Note: the first method is longer and slower. It is up to the programmer to also check for possible problems such as division by zero. Eclipse can handle these, but doesn't provide descriptive errors.

example in calling either of these routines:

```
8*0.015+:parallel_res([3,5,10,5,7])*0.015
```

Calculating the Current Given a Voltage and a Resistance

```
current:=func(voltage,resistance)
begin
  return voltage/resistance;
end;
```

example in calling:
8*:current(50,100.e3)/5.6

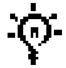
Setting Up Many Global Variables

When setting up global variables, use the prefix self then a period before the variable name. This tells Eclipse to make a permanent variable.

```
domany:=func()  // no arguments
begin
  self.variable1:=5;      // integer
  self.variable2:=62.34;  // real
  self.variable3:="Hello!";  // string
  self.variable4:=True;   // boolean - the opposite of true is Nil - not
false

  notlogical:=nil;        // tell Eclipse not to expect a number
  return "Globals set up.";
end;
```

example of calling:
:domany()



Note: since the function is returning text, you can't have any other data on the calculation line.

You can now reference any of the above variables in your calculations.

For example:
5*variable1/3+pow(5,variable1)-15.3

Converting Inches to Centimeters

```
in_to_cm:=func(inches)
begin
  return inches*2.54;
end;
```

calling:
65*:in_to_cm(5)+7*100

Calling a Program from Another Program

```
voltage:=func(resistance,current)
begin
```

```
  return resistance*current;
end;

voltage_diff:=func(voltage1,resistance,current)
begin
  return voltage1-:voltage(resistance,current);
  // :voltage calls the other procedure to calculate the other voltage
  // using the resistance and current provided.
end;
```

calling:
6-:voltage_diff(18,200.e6,5.e-6)+10

Creating a Constant

my_var:=5;

example usage:
6*my_var/4+3

creating an array of values (setup just like a constant)
my_array:=[1,2,3,4,5];

example usage:
6*:parallel_res(my_array)+7 // calling the parallel resistance program we wrote before

Sample Complex Math Functions

```
complex_add:=func(a,ai, b, bi)
begin
  a:=a+b;     // add reals
  ai:=ai+bi;    // add imaginaries
  outtext:=numberstr(a) & " + i" & numberstr(ai);  // create string
  notlogical:=nil;  // tell Eclipse to expect a string
  return (outtext); // return the string
end;

complex_mult:=func(a,ai,b,bi)
begin
  a:=sqrt(pow(a,2)*pow(b,2));
  angle:=atan(ai/a)+atan(bi/b);
  a:=a*cos(angle);
  ai:=a*sin(angle);
  outtext:=numberstr(a) & " + i" & numberstr(ai);  // create string
  notlogical:=nil;  // tell Eclipse to expect a string
  return (outtext); // return the string
end;

complex_div:=func(a,ai,b,bi)
```

```
begin
   a:=sqrt(pow(a,2)*pow(b,2));
   angle:=atan(ai/a)-atan(bi/b);
   a:=a*cos(angle);
   ai:=a*sin(angle);
   outtext:=numberstr(a) & " + i" & numberstr(ai);  // create string
   notlogical:=nil;  // tell Eclipse to expect a string
   return (outtext); // return the string
end;
```