MARTEN

Quick Start

Contents

Index

Revision 1.0

Dec. 29, 2005

Andescotia LLC Contact Information

Office:	Andescotia LLC 524 Fieldstone Dr. Bozeman, MT 59715
Website:	www.andescotia.com
Technical Support:	techsupport@andescotia.com

Contents

1

Introduction

Marten is a visual programming language designed to take advantage of principles of dataflow and object-oriented programming (OOP). Its integrated, proprietary development environment, access to OS X libraries, and thorough application framework also make it a rapid application development tool.

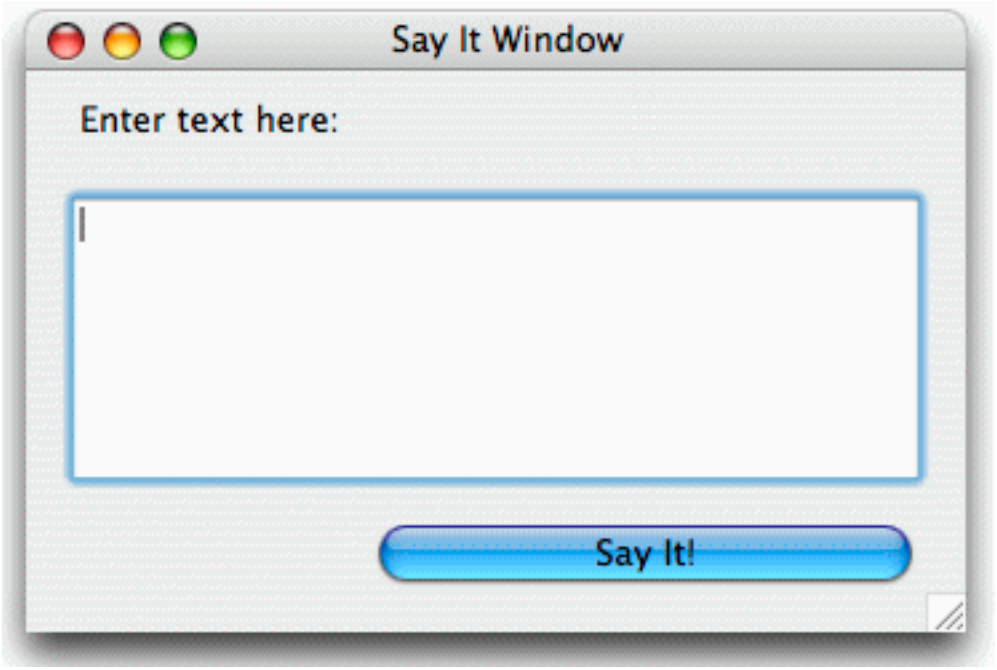
About the quick start

This tutorial introduces Marten and attempts to anticipate any questions you may have as you work through it. It takes a hands on approach; it teaches you many of the major language elements and Editor/Interpreter features by having you work through an application.

This tutorial is intended as the first step in the Marten learning curve. While experienced Macintosh programmers may use the provided examples to get started with Marten, using this tutorial is the prescribed method for most users. Once you have completed the exercises in this book, you will be ready to move on to the other titles in the Marten documentation set.

About the application

This tour walks you through the process of developing an application using Marten. After some preliminary exercises aimed at giving you the fundamentals, you will build a complete application. The final application will have your computer “speak” a text string typed by a user. The user interface will be kept simple, consisting of a **File** menu and a single window:



Users will be able to type text into the text box and when they press the **Say It!** button, the computer will "say" whatever they typed. While the process is pretty quick and painless, it touches on many aspects of the Marten development environment. This includes the following tasks:

- Creating some Marten source code using the Marten Editor
- Using with the integrated debugging features
- Using the application framework to create a user interface for the application
- Exploring how Marten can make use of code written in C.

Some Marten basics

Before you start your speech application, you'll spend time learning some Marten fundamentals. In your first exercise, you will get an understanding of the most commonly used Marten language elements and learn the basics of Marten source code storage and organization.

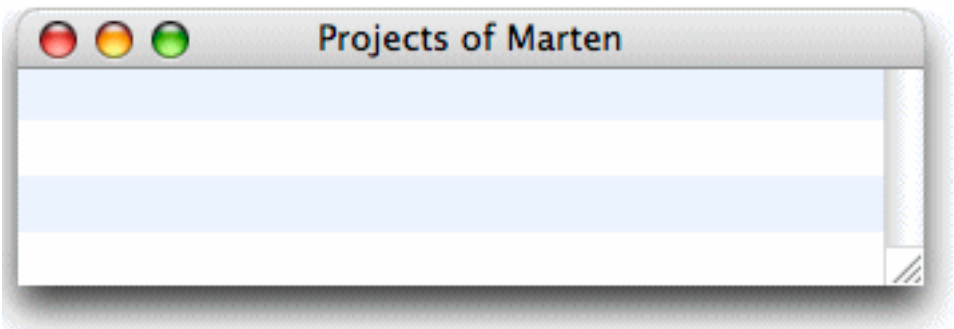
Starting Marten

Marten provides a complete development environment that integrates all aspects of the development cycle: planning, writing code, testing, and debugging.

- Locate the Marten application icon inside the Marten IDE folder.

- Double-click the icon.

  The Marten development environment starts and an empty Project window opens.



Project, library, and section basics

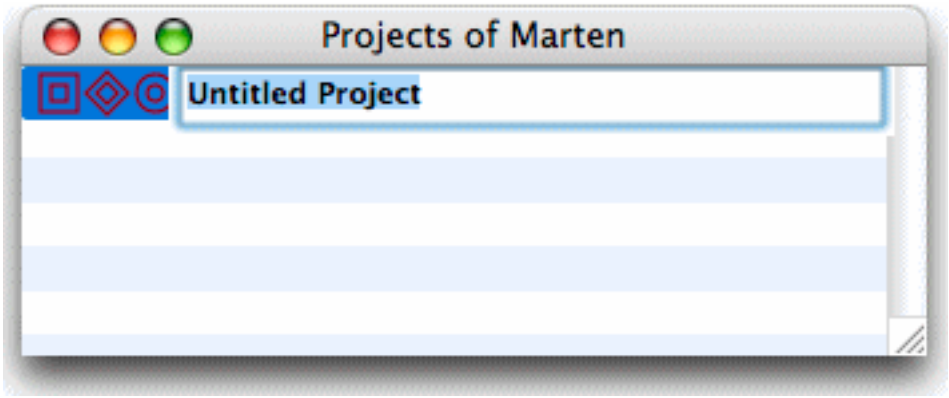
The Projects window shows currently active projects. A project provides access to the highest level elements of a Marten application. Your first task is to create a new project and new application. Marten is different from other software development environments in that you create your application first, even before defining a project.

- Command-click in any unoccupied space in the Projects window and choose **New Project** from the contextual menu.

  A Save dialog prompts you for a name for your application.

- Use the Save dialog to give your application the name **Say It.app** and create a new folder called **Tutorial Project** to store your application and source code.

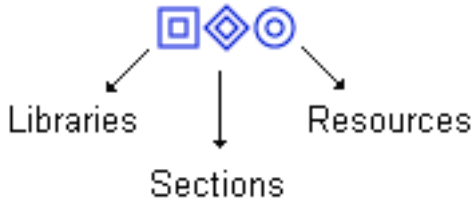
The Projects window now shows a project entry waitng to be named.



While you've already given a name to your final application, the project that organizes your application's source code is named independently.

- Type **Say It Project.vpx** and press RETURN.

The project entry icons in a Projects window represent the project elements that can be accessed from the Projects window:



Resources are generally icons and images and are used in Marten applications in the same way that they are used in other Macintosh development environments such as Xcode and Codewarrior. Libraries provide extensions to the language and as such allow access to low level building blocks such as the Carbon API, APIs to third party products, and programming interfaces to custom-written code. Sections store units of Marten source code.

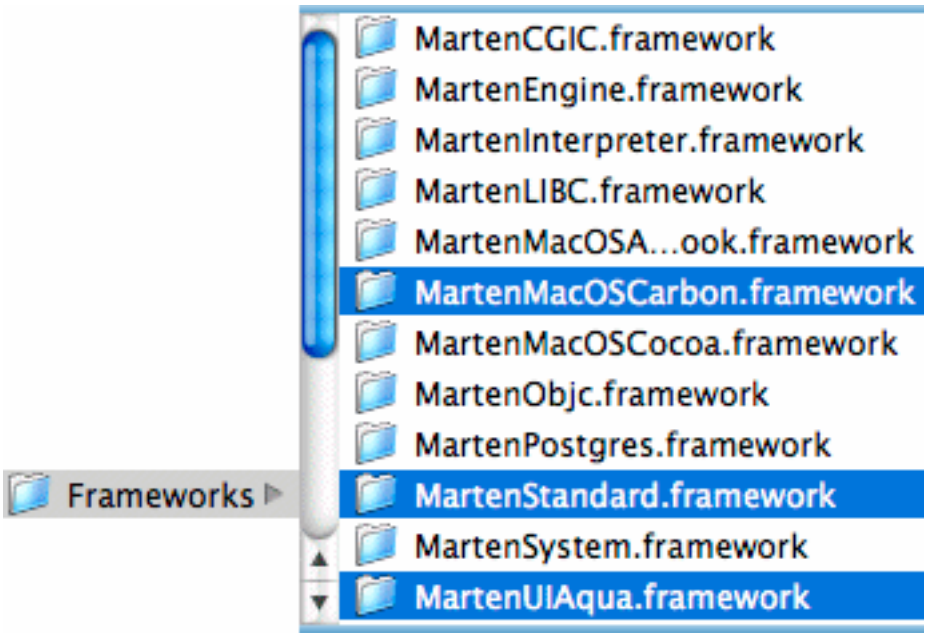
Adding libraries to your project

Next you can add some libraries to your project. In languages such as C, operators such as **+**, **&&**, and **<<** are available by default. They are part of the definition of the language. Marten, on the other hand, has very few built-in operators. Instead, additional operators (called "primitives") and other extensions to the language are added to a project as needed. These extensions are stored in libraries.

For the preliminary exercise, you will need some common programming functionality like comparison operators and math functions, as well as access to some basic Macintosh functionality such as input/output operations.

- In the Projects window, click on the **Say It Project** to select it.
- From the **File** menu, choose **Add to Project**.

  A Choose Object navigator opens.
- Navigate to the **Frameworks** subfolder of the **Marten** folder and add **MartenMacOSCarbon.framework**, **MartenStandard.framework**, and **MartenUIAqua.framework**.

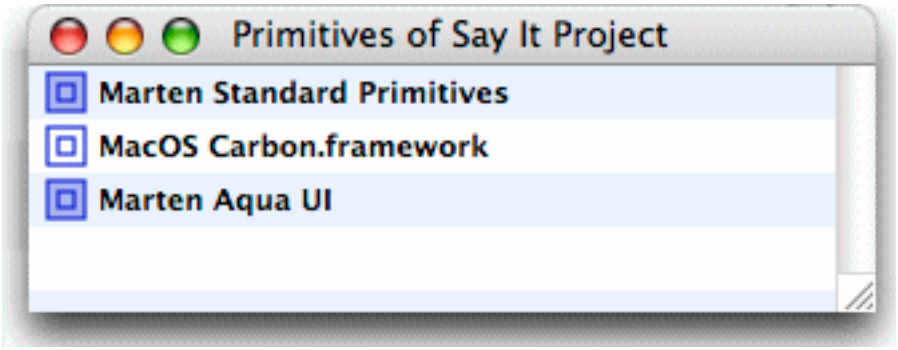


- Click **Choose.**

The Libraries icon in the Projects window takes on a different, "filled in" appearance to indicate that libraries are now present in the project.

- Double-click on the libraries icon.

  A **Primitives of SayIt Project** window opens.



This window shows you the collections of primitives you just added. You'll learn more about primitives shortly. For now, while you could drill down deeper to view lower level components, your project is complete with regard to libraries.

- Click the close box in the **Primitives of Say It Project** window.

Adding a section to your project

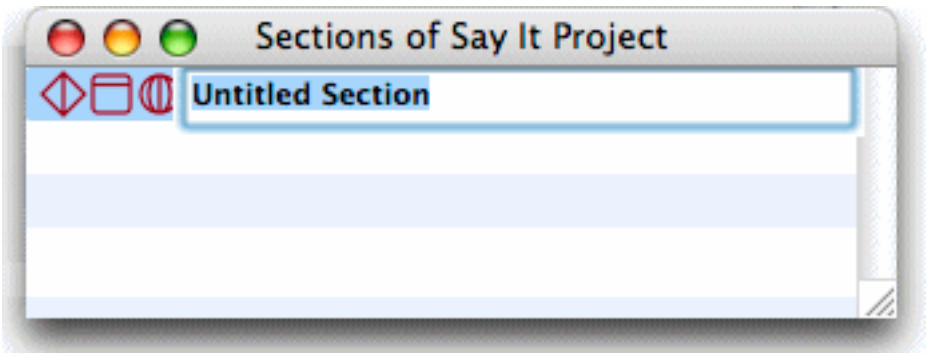
Next, you need to create a Marten section. A section is a file on disk, used to store a "unit" of Marten source code. Sections are typically used to group functionally-related code, a set of classes that implements a radio button, for example. In addition to being a useful organizational tool, sections allow you to reuse code components in multiple projects.

- Double-click the sections icon in the Projects window.

An empty **Sections of Say It Project** window opens.

- COMMAND-click in any unoccupied space in the Sections window.

A new section is created, ready to be named.



- Type **Say It Section** and press RETURN.

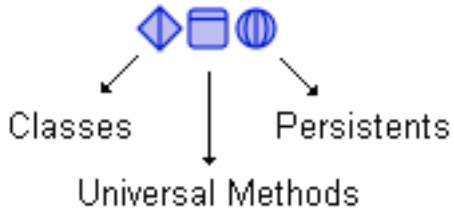
Now you've named your section but you haven't saved it. Like a project, a section is a disk file that must be saved.

- With your new section selected, from the **File** menu, choose **Save**.

  A Save dialog opens.

- Name your section **Say It Section.vpl** and save it in the same location where you stored your project.

With some of the bookkeeping out of the way, you're ready to start on some Marten programming. In the Sections window, each section entry has three associated icons:



Each icon represents a type of Marten element that can be contained in a section:

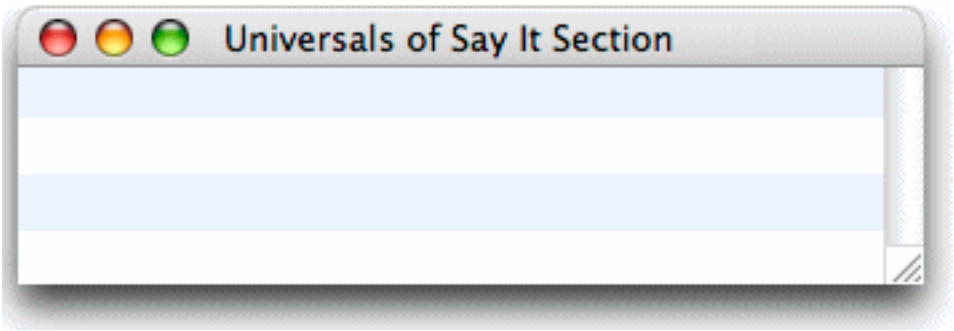
- *Persistents* store user-defined data
- Methods contain cases, which in turn contain code. You will be working with *universal methods* shortly
- *Classes* contain *attributes* and *class methods*

Working with universal methods

You will not be working with classes until you are ready to put a front end on your application. For now, you will be working with universal methods, general-purpose methods not associated with particular classes.

- Double-click on the Universal Methods icon in the **Sections of Say It Project** window.

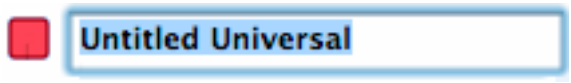
- An empty **Universals of Say It Section** window opens.



A Universal Methods window displays all the universal methods contained in a section.

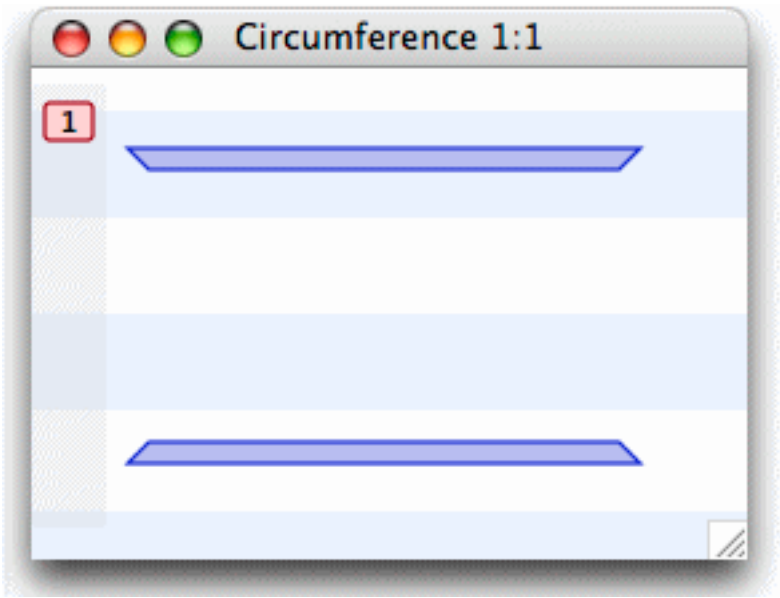
- COMMAND-click anywhere in empty space in the Universal Methods window.

  An untitled universal method icon is created.



- Type **Circumference** and press RETURN to name the new method.
- Double-click on the **Circumference** icon.

  A Case window opens.



Methods, both universal methods and class methods, are composed of one or more *cases*. A case of a method is analogous to a case of the case or switch statement in other programming languages. Marten provides logic for moving from case to case of a method. In fact, the case mechanism is a large part of control flow in Marten.

The Case window currently is empty with the exception of two horizontal bars: the *input bar* and *output bar* operations which provide the method with inputs and outputs.

What goes on in methods

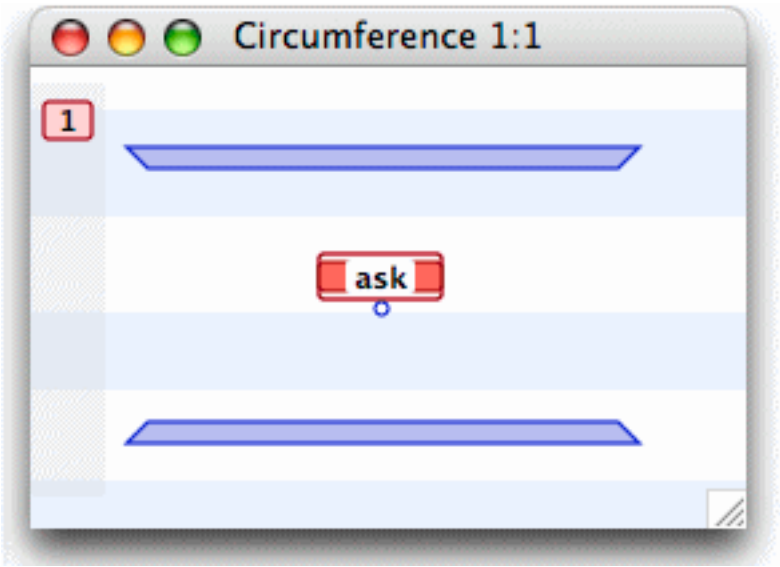
Ultimately, your application is going to take a character string as an input and have your computer "speak" that character string. Before working toward that functionality, though, it is worthwhile to take a look at what goes on inside a method. In doing so, you will create a method that calculates the circumference of a circle, given the radius.

- While holding down the COMMAND key, click about an inch below the input bar and toward the left of the window.

  An untitled *operation* is displayed.

- Type **ask** and press RETURN.

  The appearance of the operation changes.



What are operations?

Operations are the basic unit of execution in Marten. They perform some action. For example, an operation could call a primitive, return the value of an attribute, or call a method. There are a number of different types of operations in Marten and you will see more of them as you proceed.

In this case, when you typed **ask**, Marten recognized the name of a primitive loaded when you added libraries, earlier.

The **ask** primitive prompts the user for input which can then be passed to another operation. The node at the bottom of the operation is an output *root*; By default, the **ask** primitive takes no inputs and returns a single output: what the user typed.

In this method, the **ask** primitive will be used to prompt the user for the value of the radius.

- Once again COMMAND-click, this time below and to the right of the **ask** primitive.

  Another operation is created.

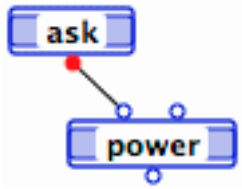
- Type **power** and press RETURN.

The **power** primitive performs exponentiation. It takes two inputs, the number and the exponent. This is reflected by the fact that the operation now has two input *terminals*. The **power** primitive returns a single output, the result.

You are beginning to see the purely visual aspect of Marten. While other "visual" development environments take a visual approach for higher level aspects such as user interface design, Marten is visual at the lowest levels.

- Click on the **ask** primitive's output root to select it.
- While holding down the OPTION key, move the pointer directly over the **power** primitive's leftmost input terminal and click.

  A **datalink** is created.



What is a datalink?

Marten is a *dataflow language*. In a typical procedural language, variables are declared and the values of these variables are manipulated by sets of instructions. In Marten, data passes from operation to operation via datalinks. The output of the **ask** primitive will be passed as an argument to the **power** primitive.

- Double-click the power primitive's rightmost terminal.

  A constant with an initial value of NULL is created.

Different types of operations have different icons. Up until now you have been typing in only names which Marten recognized as primitives. When Marten recognizes a name as a primitive, the icon is automatically changed from a simple icon to a primitive icon. As you work through the Marten learning curve, you will learn more about different icons and their meanings by using the options available through the **Operations** menu.

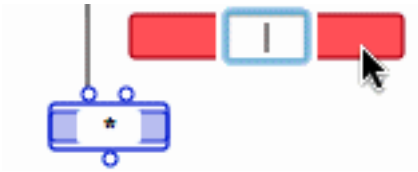
- Type **2** and press RETURN.

Remember: the circumference of a circle = **pi** * $\mathbf{r}^2$

- COMMAND-click, this time below and to the right of the **power** primitive then type * and press RETURN.

  This creates a call to a primitive that performs multiplication.

- Click on the **power** primitive's root to select it then OPTION-click directly over the * primitive's leftmost terminal to create a datalink between the two operations.
- COMMAND-click above and to the right of the * primitive.



- Type **pi**.

  The Simple operation is changed to a Primitive operation.

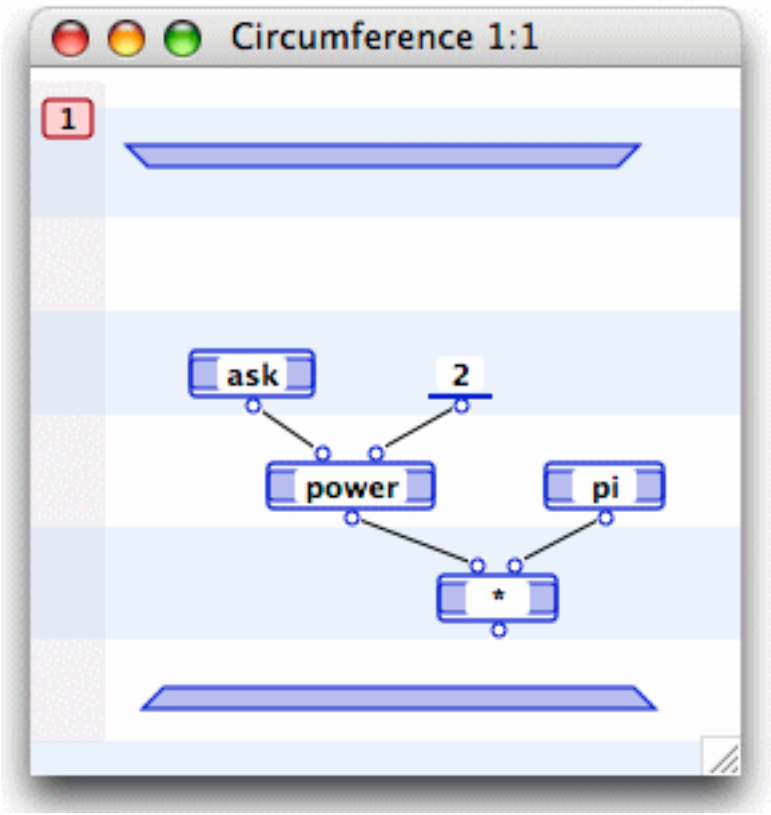
You could have once again used a constant to pass the value but Marten has a primitive that returns the value of pi.

- Create a datalink from the **pi** primitive to the * primitive's rightmost terminal.

Due to the default size of a Case window, the window can get cluttered pretty quickly. This is easily remedied by making the window bigger and moving your operations to make the method more "readable."

- Click drag the grow box down and to the right to open up the window about an inch wider and longer.
- Click-drag the output bar toward the bottom of the window to free up some space for more operations.

- Rearrange the other operations in this case so that it looks like the following diagram



Note that as you move operations around the case window, all terminals, roots, and datalinks remain intact.

If you were to execute your method right now, a generic dialog box would prompt you for the value of the radius. A minor customization can be achieved with a single operation.

- Move the cursor to a location just slightly above the **ask** primitive.

- COMMAND-click there to create another terminal:

While some primitives have a fixed *arity*, the number of inputs and outputs, others allow you to create more terminals and roots.

Hints & Tips

Should you accidently create an operation and wish to delete it:

- Click once outside the operation to disable text editing.
- Click the operation once and press DELETE.

- Double-click on the new terminal.

  This is a shortcut that creates an attached constant with an initial value of NULL.

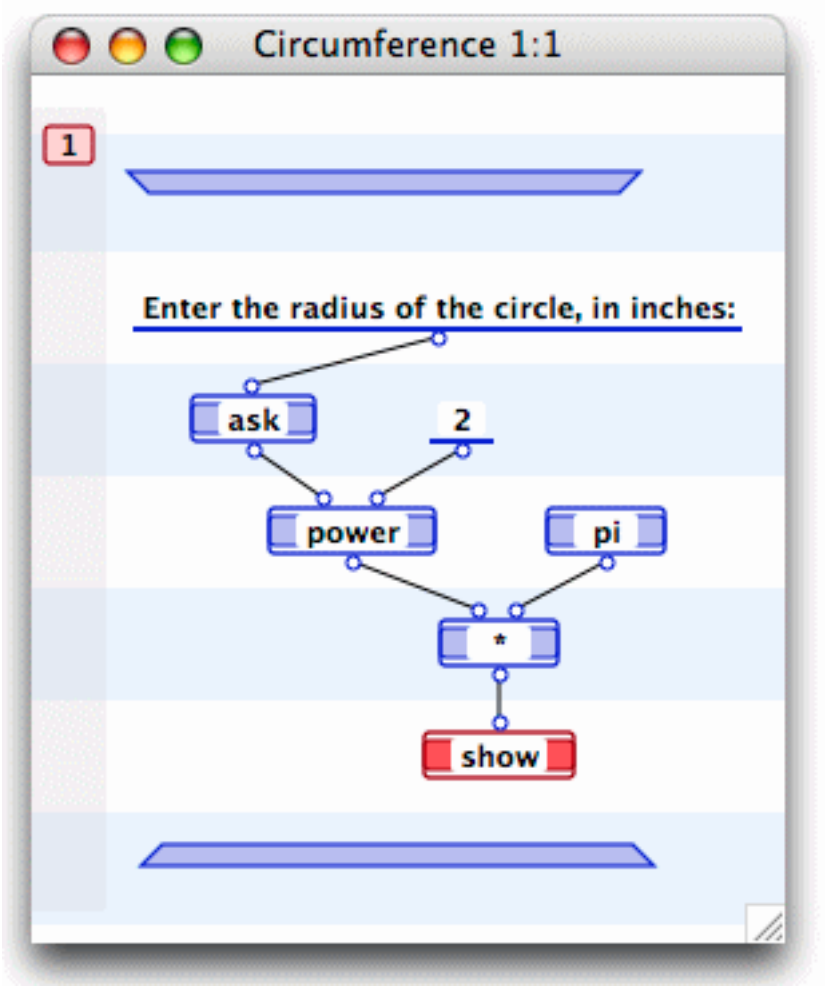
- Type **Enter the radius of the circle, in inches:** and press RETURN.

This provides a minimal front end for your input dialog. Later on, you will see how Marten allows you to quickly and easily build a sophisticated graphical user interface. Presumably, you would want to customize the output of your method, but for now just assume that the value of the **pi * (r**2)** calculation returns a value expressed in square inches. With that in mind, the output of the method can be handled with a single operation.

- Create a new operation below the * primitive, name it **show**, and place a datalink between the * primitive's output root and the **show** primitive's input terminal.

  The **show** primitive will display the result of the calculation.

By now your method should look something like the following:



3

Using debugging and testing features

You are now ready to execute this bare bones method. In this chapter you learn will how Marten lets you execute your code in interpreted mode.

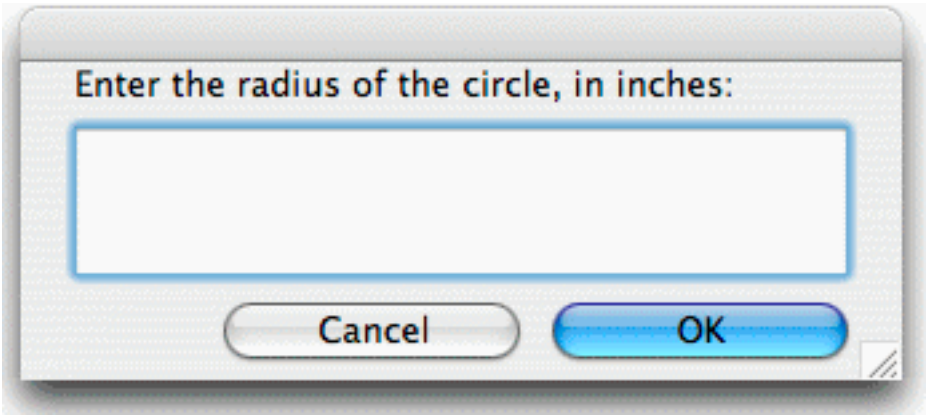
Executing your method

When you started this exercise, in addition to providing names for the project and section, you provided a name for the Project application. The Project application runs as a process separate from the Editor environment and it has two key functions. It lets you run methods or applications in interpreted mode and when updated properly, it allows your project to run as a stand-alone Macintosh application.

For now, you will look at some of the interpreted code features.

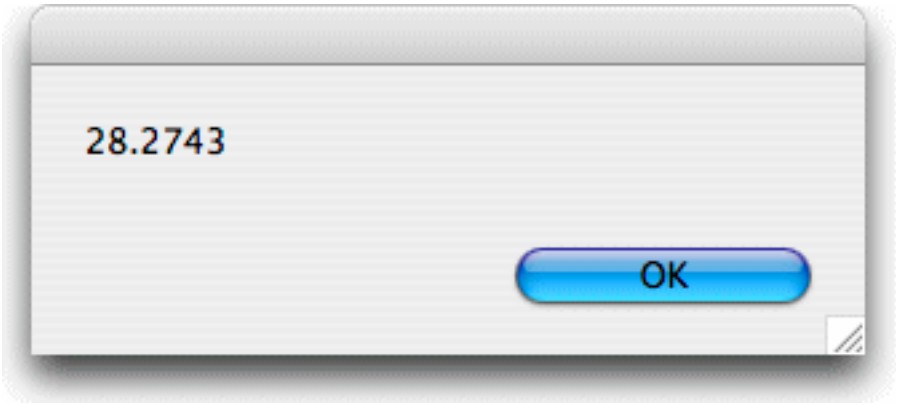
- With your **Circumference** case window frontmost, from the **Run** menu choose **Run Method**.

  A dialog box is displayed prompting you to enter a value.



- Type **3** and click the **OK** button.

  The result is displayed.



- Click **OK** to close the window and finish execution of the method.

The integrated debugging features put all testing at your fingertips. You can test individual methods, as you just did, or entire applications. The Interpreter also has many debugging features, some of which you will see shortly.

Adding error processing to your method

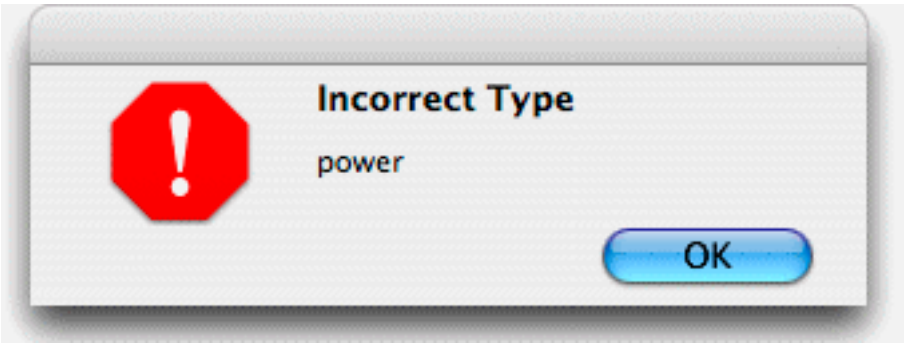
If you were writing a final application, you would want to add some enhancements. Depending on requirements, this might mean as little as providing more information in the two dialog boxes or as much as creating a sophisticated graphical user interface for the application. For now, you will only add some minor error handling.

Forcing an error

As your method currently works, if an end user entered anything other than a number at the prompt, it would incur a fault or error. To demonstrate:.

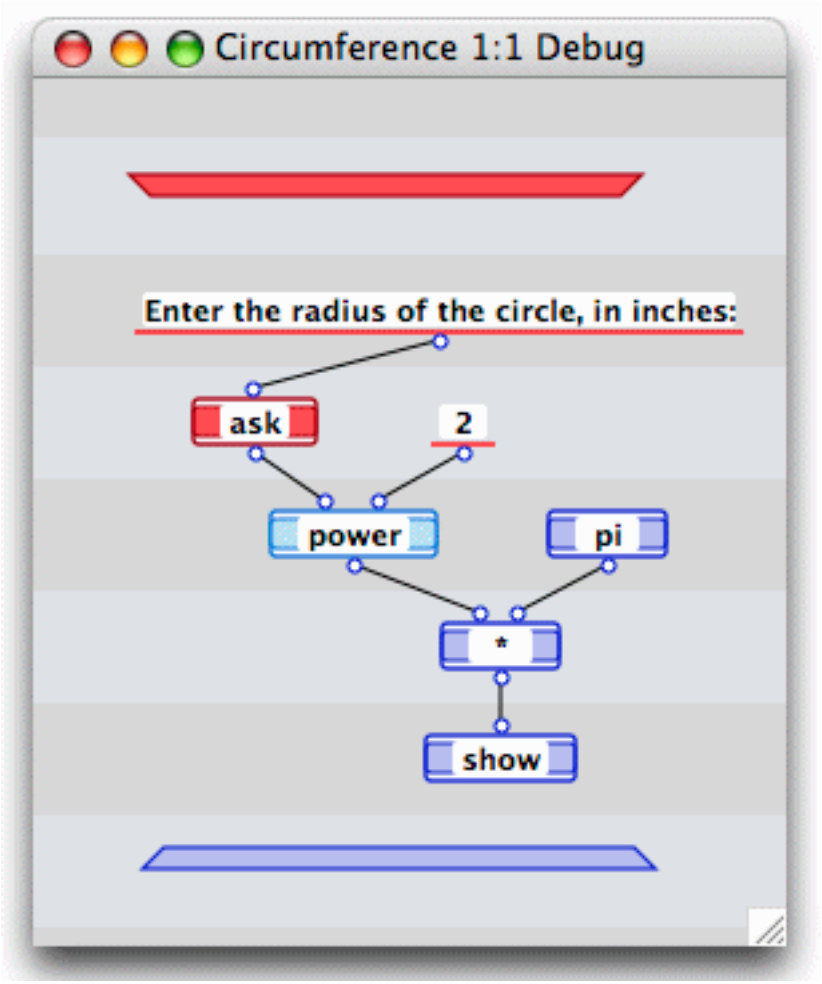
- Execute the method again and at the prompt for a number press the RETURN key.

  The following error message is displayed:



- Press **OK**.

  You are presented with an execution window, distinguishable from the Case window by the darker background background.

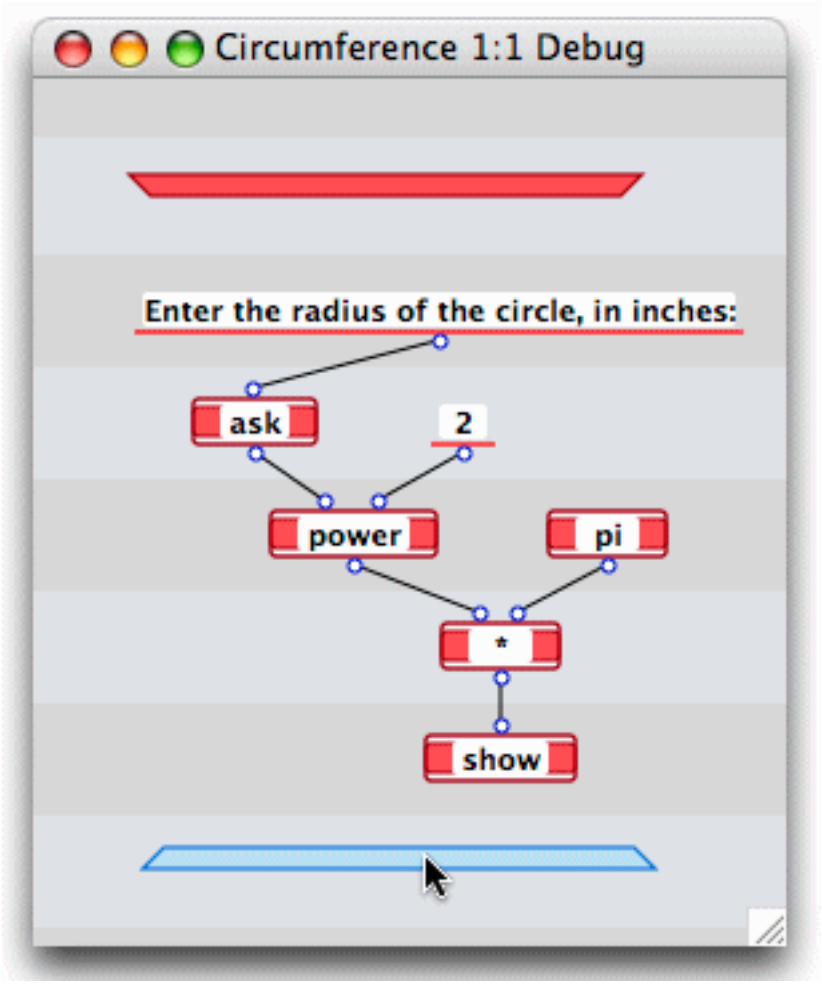


When the Project application detects a breakpoint or an execution error, it suspends execution. While you will learn more about breakpoints later on, for now you can concentrate on execution errors.

When the Project application suspends execution, you have access to Marten's debugging features. The Execution window, for example, lets you toggle back and forth between the Editor environment and execution of your code, and provides options such as operation-by-operation execution, rollback and roll forward options, and a variety of ways of stepping through the code.

In order to fix the current problem, it is first necessary to terminate execution of this method. First, in order to bypass the execution eror on the **power** primitive, you can roll forward execution to the output bar.

- COMMAND-click on the output bar



The highlighting shifts from the **power** operation to the output bar. Like other operations in a Case window, the output bar is executable.

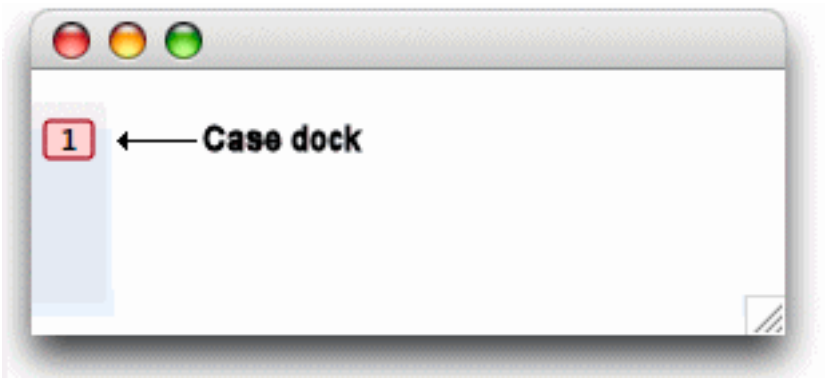
- Press RETURN to execute the output bar.

  This comples execution of the method, closes the execution window, and returns you to the **Circumference** case window.

About cases

As a fix to this situation, you will add a test that ensures a number was entered. If the test fails, you will pass control to a second case of your method, a case that does nothing more than display an error message.

At the left of the Method window, is the Case dock.



The Case dock lets you create new cases, display the different cases of a method, select cases for operations such as deleting, and reorder the cases within a method. The **Circumference** method currently has only one case:

- COMMAND-click in the Case dock to create a new case.

  The Case dock is updated to show a second case.

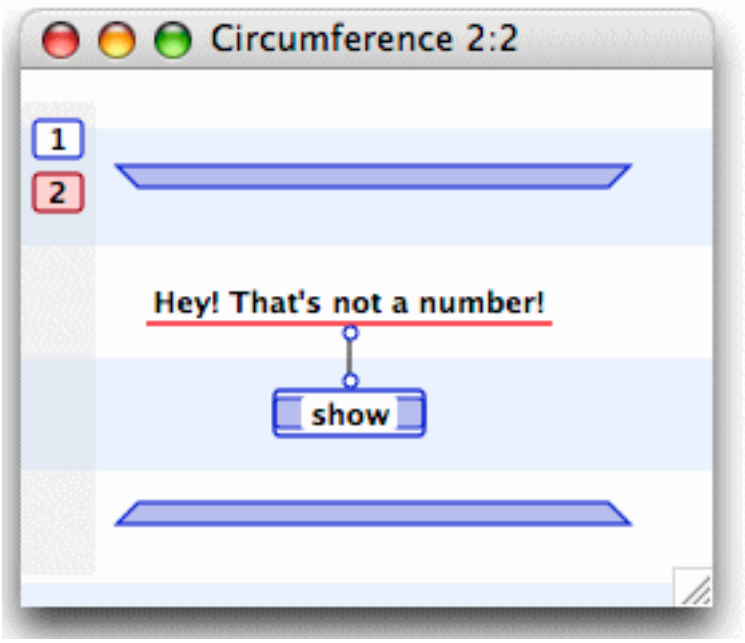
- Double-click the new (**2**) icon in the Case dock..

  A Case window for the new case opens.

This case will be trivial. Its only task will be to display an error message. It will consist of two operations: a constant to store the message and a call to the **show** primitive to display the message.

- Make the second case of the method look like the following diagram.

  The constant's value should be **Hey! That's not a number.**.

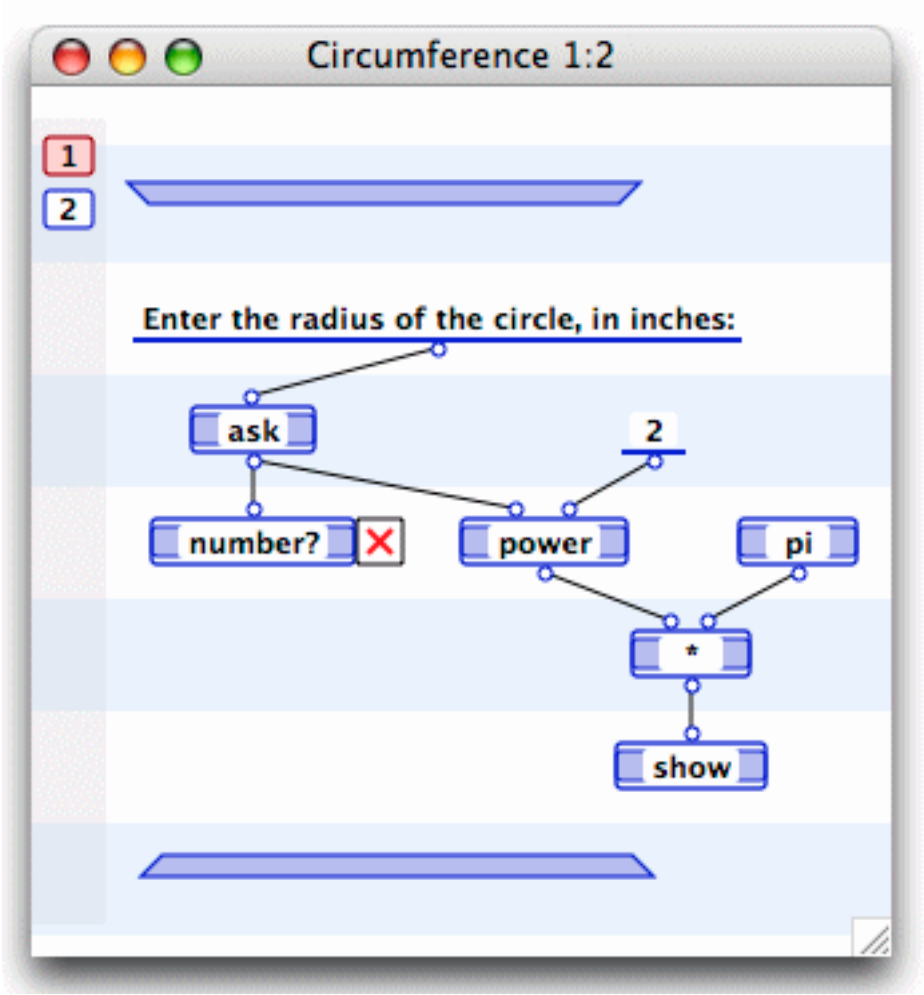


- Click the left arrow button.

  The case containing your calculation is displayed frontmost.

The only remaining thing to do is to ensure that the input is tested and if it is not a number to pass control to the error handling case.

- Below the **ask** primitive create a new operation that calls the **number?** primitive and create a datalink from the **ask**'s operation's output root to the new primitive's input terminal.



The **number?** primitive tests whether its one input is a number. The little box with the **x** inside is a *Control*. Controls are used to aid control flow in a Marten application. In this case, a Next Case control is attached to the **number?** operation. If the check for a number fails, control is passed to the next case of the method.

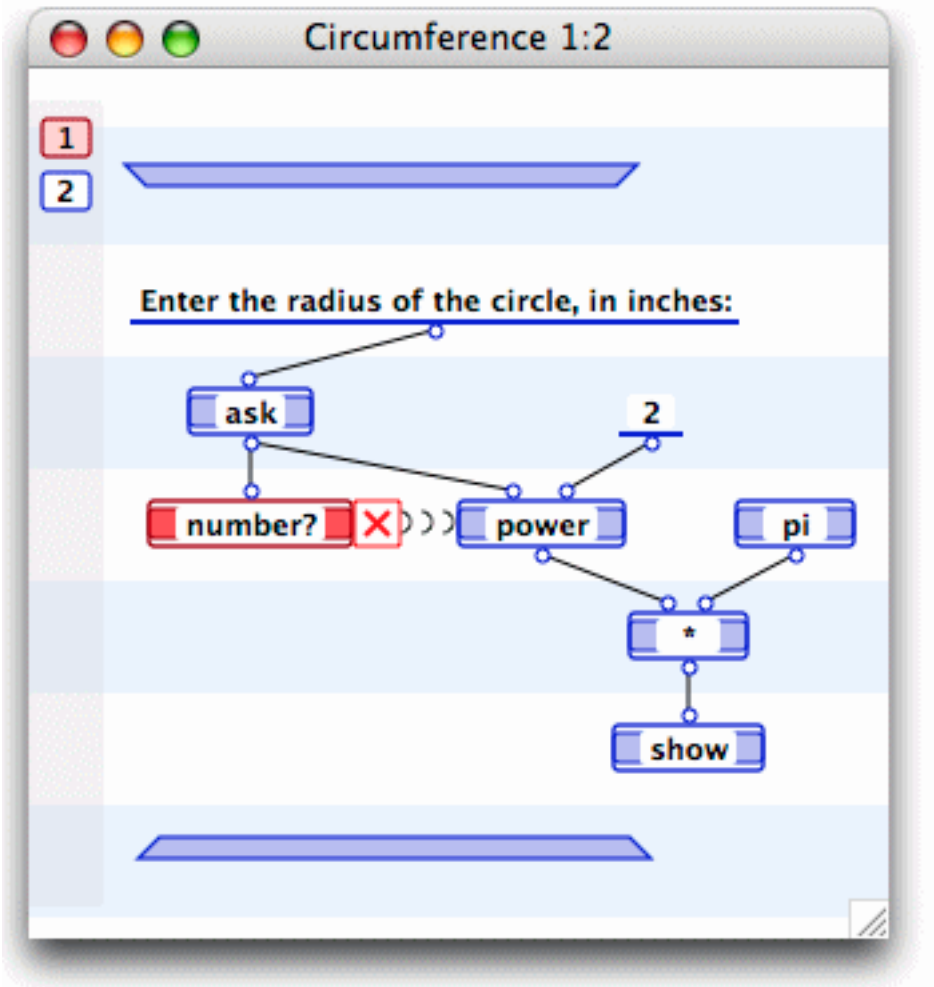
- Click the **Controls** menu to take a look at the items in the menu.

The menu items in the **Controls** menu provide all the control flow functionality you need to write a Marten application, from case navigation to looping mechanisms. You'll learn more about controls as you progress throughthe Marten learning curve.

Controlling sequence of execution

Within a case, an operation will execute as soon as all inputs have arrived on its input terminals. This would be a problem with the **Circumference** method since you want to check the input before performing the calculation.

- Click on the left side of the **number?** operation, anywhere outside the text area, to select it.
- While holding down the OPTION key, click on the **power** primitive.



You've just created a *synchro*. Synchros control the sequence of execution of the operations within a case; in this case you've just ensured that the **number?** primitive will be executed before the **power** primitive. Since operations can only execute when

all inputs are available, you know that the **ask** primitive will be executed before the **number?** primitive.

Looking at a debugging tool

As mentioned earlier, in addition to straight editing capability, Marten lets you run code interpreted and provides debugging and testing features. It's time to see some of this functionality in action.

- From the **Run** menu choose **Debug Method**.

This command opens an Execution window on the method letting you execute the method "manually;" one operation at a time. Walking through the code in this manner lets you see exactly what is going on in a method as it executes. When the Execution window opens, the input bar is highlighted, indicating that it is about to execute.

The input bar is executable as are other operations; if your method takes input parameters, executing the input bar passes the parameters into the method. Likewise, a constant operation executes; its job is to pass a value to another operation.

Earlier, you simply used the RETURN key to execute an operation. Now it's time to try a different method.

- From the **Run** menu, choose **Step**.

  Highlighting shifts to the constant operation, indicating that the input bar has executed and the constant is next to execute.

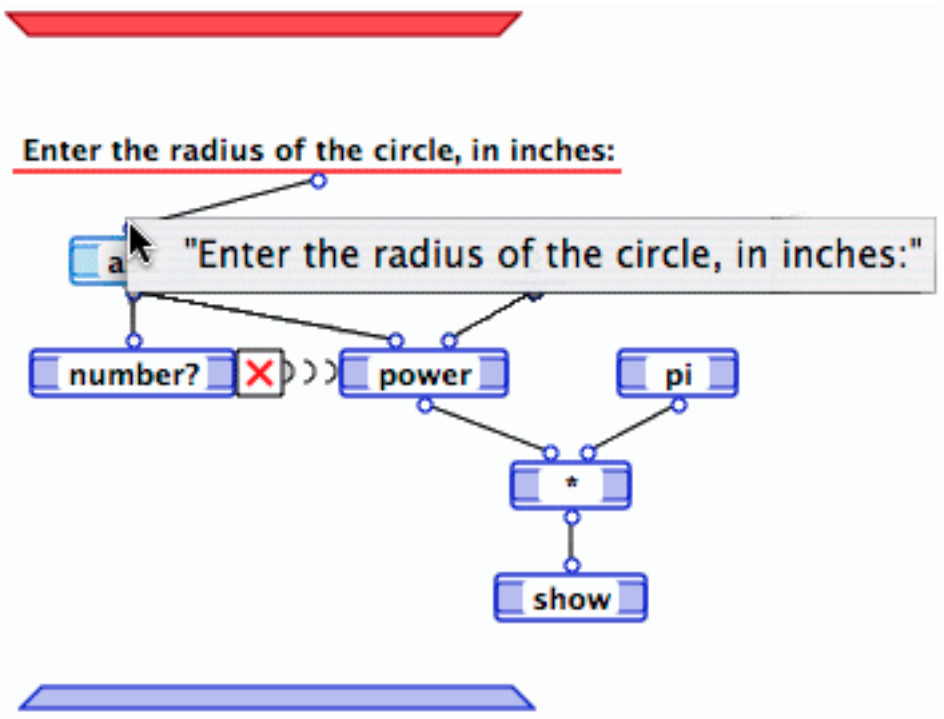
In addition to the **Step** command that executes the next operation, the **Run** menu provides a number of options for stepping through your code. You can:

Step In	To a called method to open the case window of that method for execution.
Step Out	To return to a calling method to resume execution of that method.
Step Over	To skip execution of the next operation to execute.

You'll learn more about these options as you learn more about Marten.

- Press RETURN to execute the constant operation.
- Now, place the cursor directly over the input terminal of the **ask** primitive and click the mouse button leaving it depressed for a few seconds.

  The value that has arrived on the **ask** primitive's terminal is displayed.



You can well imagine the power of a debugging feature like this. You can inspect values of the data flowing through a method's datalinks. And giving you more debugging power, particularly in the case of execution errors, is the ability to actually change values on the fly. For now though, you are simply going to finish execution of your method.

- Press RETURN to execute the **ask** primitive.

  The input dialog for your method is displayed.

- Type **f** to test if your error processing is working properly then press RETURN.

  The Execution window shows that the **number?** primitive is about to execute.

- Press RETURN to execute the **number?** primitive.

  An Execution window opens for the second case of your method.

- Press RETURN three times and watch as the method executes the remaining operations and then displays the message indicating that the input was not a number.

- Keep pressing RETURN until the execution terminates, taking you back to the Case window for the method.

Single step execution and the ability to display values during execution are just two of the debugging techniques provided by Marten. You'll see many more as you become familiar with the Marten development environment.

Starting your speech application

With some Marten fundamentals under your belt, you're ready to start your speech application.

What are external procedures

You can start by creating a new universal method.

- In the **Universals of SayIt** window, COMMAND-click to create a new universal method, name it **Speak-Text**, then double-click on the new universal to open it.
- COMMAND-click in the lower part of the Case window to create an operation and name it **SpeakString**.

  The icon changes slightly to indicate that it is a call to an external procedure.

External procedures are calls to functions and methods written in C. This is the mechanism by which applications written in Marten make use of public interfaces such as the Macintosh Carbon API or the APIs for third-party software.

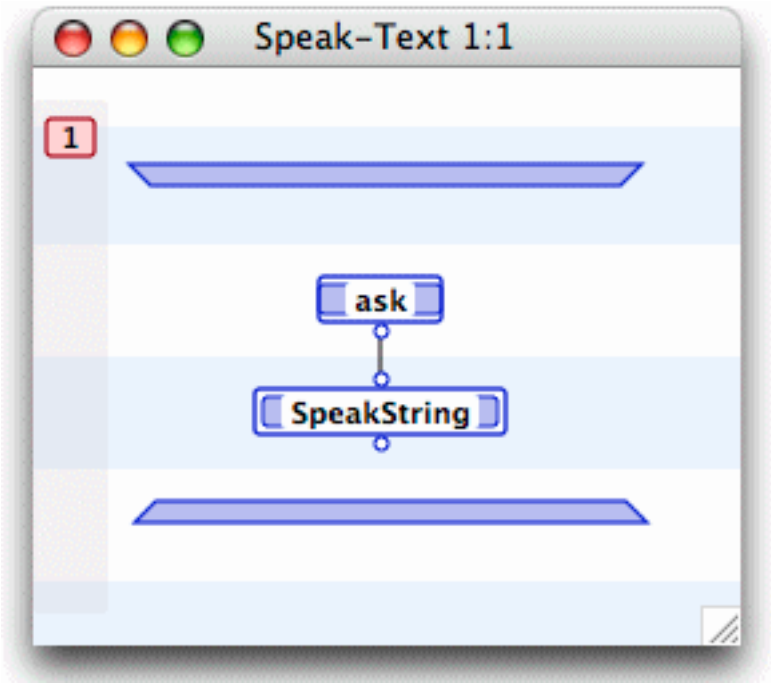
Like the primitives that you loaded earlier in this exercise, external procedures are stored in libraries. A set of commonly used external procedures is packaged with Marten and you can also create your own. You create "dictionary" entries for external procedures, constants, and structures.

Note: In addition to calling third party code from Marten, you can also write your own primitives.

Ultimately, users of your application will provide a character string as input. Later on, you'll build a graphical user interface that passes the text to a method. For now, to simplify the development process, you'll use the **ask** primitive again to prompt for a character string.

- Near the input bar, create an operation that calls the **ask** primitive and connect its output root to the input terminal of the **SpeakString** external procedure.

  Your method should now look like this:



- From the **Run** menu choose **Run Method**.

  A dialog is displayed prompting for input.

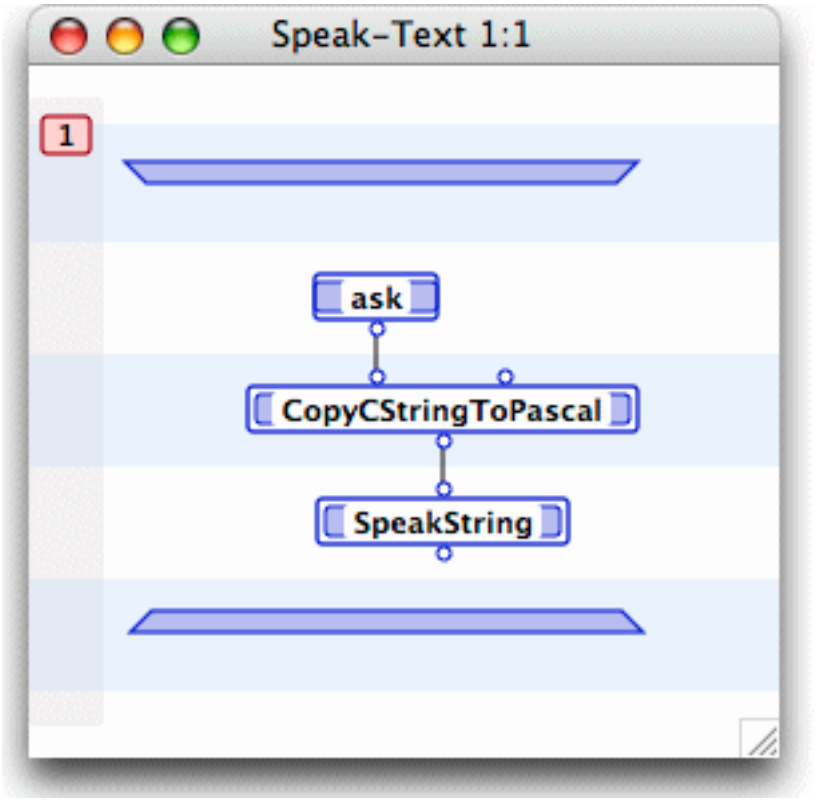
- Type **Hello there, Malcolm.** and press RETURN.

While this executes properly from Marten's perspective, you'll notice that the speech produced is not exectly what you would expect. Forgetting low-level details for the moment, the problem is with the format of the string delivered by Marten. Specifically, SpeakString, the Macintosh Carbon API routine that does the work of converting the text to speech, expects a pointer to a string formatted as a Pascal string rather than a C string.

Finalizing your method

This is easily remedied by tweaking the input to the **SpeakString** external procedure. In fact, there is another external procedure that will convert the provided string appropriately.

- In the **Speak-Text** method window, click on the **ask** primitive's root to select it, then, while holding down the OPTION key, click on the **SpeakString**'s terminal to delete the datalink.
- Create a new operation between the two operations and name it **CopyCStringToPascal**.
- Create datalinks between the operations so that the method looks like the following:



Now you can properly play the text typed.

➤ Again, from the **Run** menu, choose **Run Method**.

The text you typed is played for you!

Congratulations! You've built and executed a method that converts text to speech. This has been a simple process, showing you some of the fundamentals of Marten and giving you an idea of its power as a development environment.

Now you can take this a step further and build the user interface that will let a user type text, push a button, and have the words spoken.

Loading a framework

Marten ships with a number of application frameworks. Each framework consists of a thorough set of classes aimed at particular development paradigms. Each framework includes a set of editors that make it easy to make use of these classes and structure development of your applications. The frameworks are all Marten native code.

In this chapter you will add the Small Application Classes (**Small App**) Framework to your project. This will provide the materials you turn your Speak-Text method into a fully functioning Macintosh application with a slick, graphical front end. It will also give you a chance to see Marten's support for object-oriented programming.

Note: Development of the Small Application Classes framework is ongoing and the following screenshots may differ slightly from the current shipping version.

Loading the Small App Framework

The Small App Framework is contained in a set of sections you can add to your project.

Sections are individual disk files that store source code. Entire applications are held together with a project file that contains a listing of all the sections included in the application.

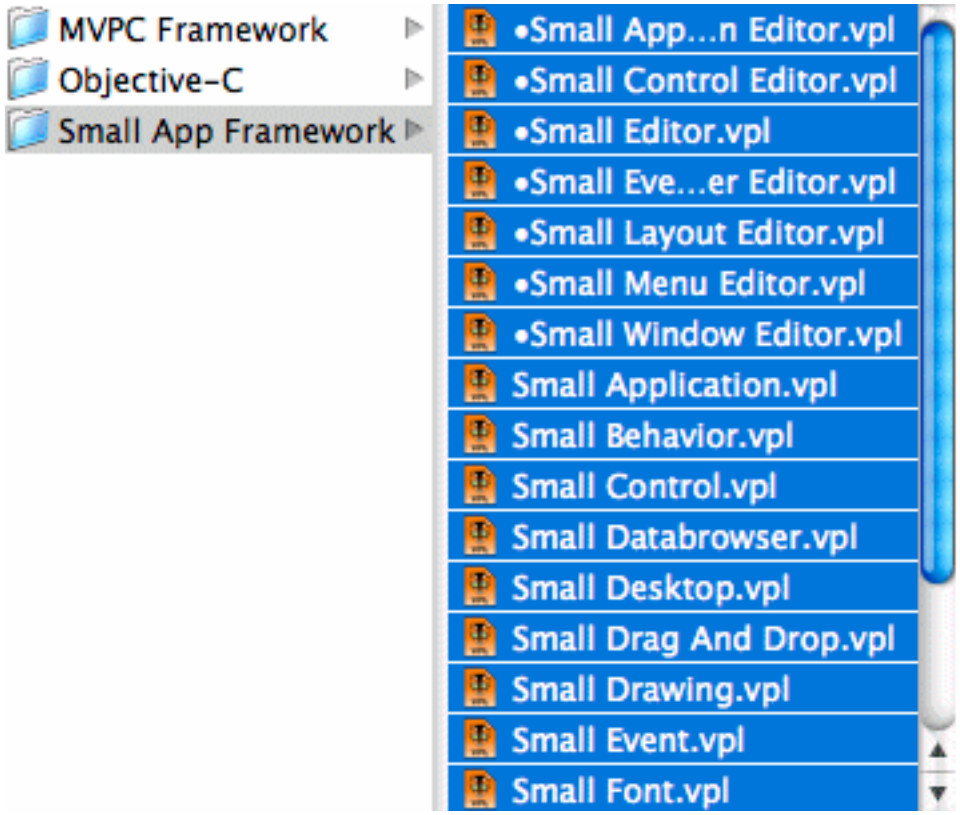
- From the **File** menu, choose **Add To Project**.

  An Choose Object dialog opens.

- Locate the **Small App Framework** folder.

  By default, it is installed in the **/Library/Marten** folder

- Select all files in that folder.



➤ Click **Choose**.

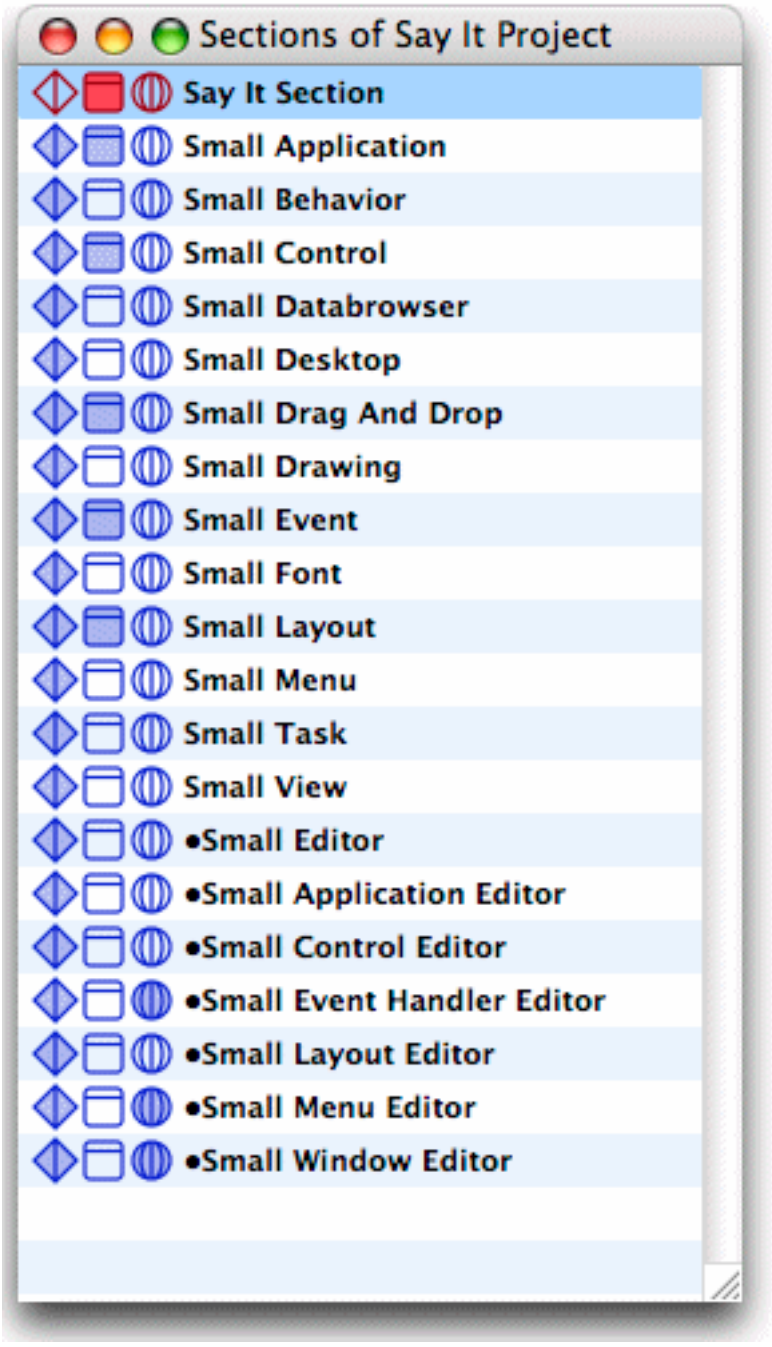
Working with the sections added to our project

Your **Sections of Say It Project** window is updated with the new components. Each section contains a functionally-related "unit" of code consisting of classes, univeral methods, and persistents.

When you first started this tutorial you created a new section. This time, you added existing sections to your project. As you get more experienced with Marten you will use the sections mechanism to reuse commonly used code.

Most people have a preference for the order of sections. You can reorder your sections by dragging a section up or down to a new location.

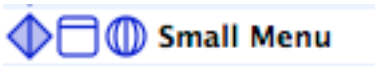
➤ Reorder your sections so that your Sections window appears as follows:



Note: Each **... Editor** section implements a visual editor for working with instances of the associated class. You'll work with these editors shortly.

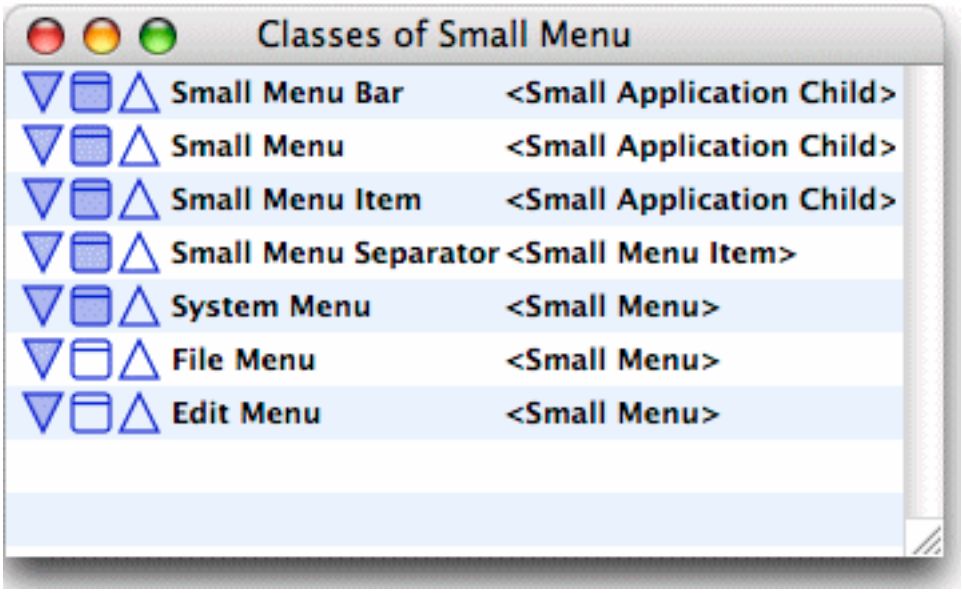
About classes

You've already seen universal methods as components of a section. Now it's time to take a look at another section component: classes. Remember: each section entry has icons representing the section's classes, universal methods, and persistents.



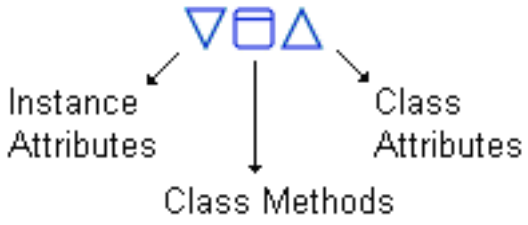
- Double click on the **Small Menu** section's class icon.

A **Classes of Small Menu** window opens.



This is the Class window, another of the Marten development environment's native editors. The Class window lets you work with the classes in a section. Typically, a section will contain a set of related classes.

As in all OOP implementations, classes are composed of methods and attributes. Attributes are further broken down into Instance attributes and Class attributes. This is reflected in the set of icons representing each class.



Unlike some other OOP implementations in which an existing programming language is beefed up with class support, Marten was designed for object-oriented programming. Classes, methods, and attributes are integral aspects of Marten.

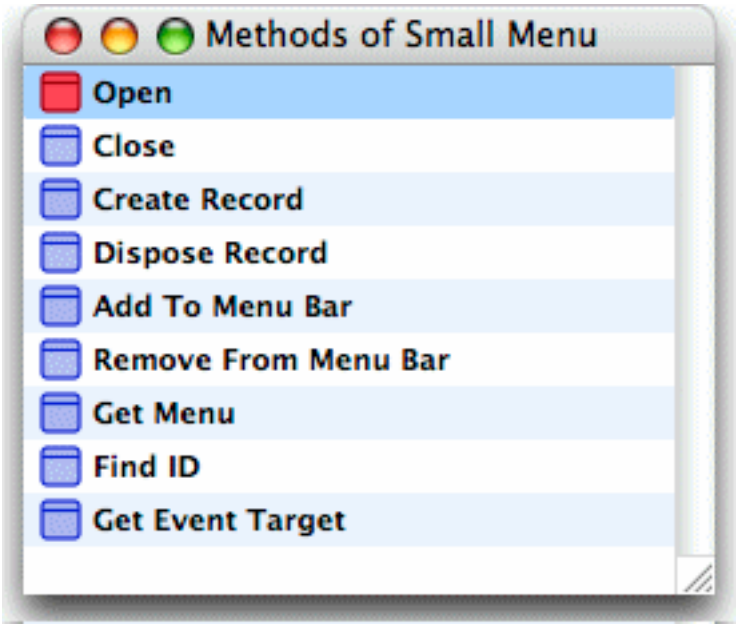
- Double-click the Instance attribues icon of the **Small Menu** class.

  An **Instance Attributes of Small Menu** window opens.

This class is used to create and programmatically manipulate instances of a standard Macintosh application menu. Its attributes store the properties of a typical Mac menu: the items appearing in the menubar, and so on.

- Close the Instance Attributes window then double-click the Class Methods icon of the **Small Menu** class.

  A **Methods of Small Menu** window opens.

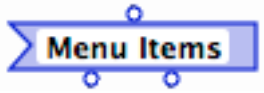


The methods of the **Small Menu** class provide all the basic functionality required to work with instances of that class. The methods whose names start with **Get**, for example, return the values of attributes.

- Double-click on the method icon labelled **Open**.

  The first case of the method opens.

Note the **Menu Items** operation. Once again, this is an operation you haven't seen before.



It's a Get operation. Given an instance as input, it returns the value of the attribute indicated by its name. It also returns the instance. While the use of Get in the method name is a simple convention, Get operations, like their counterparts Set operations, are parts of the Marten language. You'll get more familiar with class-specific language components of the language as you work more with Marten.

- Close the **Open**, **Methods of Small Menu**, and **Classes of Small Menu** windows.

6

Building the user interface

In this chapter you will use the Small App Framework sections to turn your Speak-Text method into a Macintosh application.

Key properties of your application

Your first task is to create an application. The framework you loaded has an application class named **Small Application** that has many attributes that your speech application will need. You can create a new class and then make that class a subclass of the provided application class.

- In the **Sections of Say It Project** window, double-click the classes icon of the **Say It Section**.

A **Classes of Say It Section** window opens.

This window is currently empty since you haven't created classes for this section yet.

- Command-click in the Classes window to create a new class.
- Name the class **Say It Application**.

Now you can designate this class as a subclass of the **Small Application** class. Marten makes this very easy.

- Holding down the CONTROL key, click on the new class, and then from the context menu choose **Parent Class**, and then choose **Small Application**.

The **Say It Application** class item is updated to indicate that its parent class is **Small Application**.

Creating an application instance

Now that you have an application class, you can create an instance of the application. You'll create this using a Marten persistent. A persistent stores data ranging from simple data types to complex class instances. It is similar to a global variable in other languages. Range of use of persistents is varied as well. In addition to storing instance values, you can use persistents to store values between interpreted executions of your code.

- In the **Sections of Say It Project** window, double-click the persistents icon of the **Say It Section**.

An empty **Persistents of Say It Section** window opens.

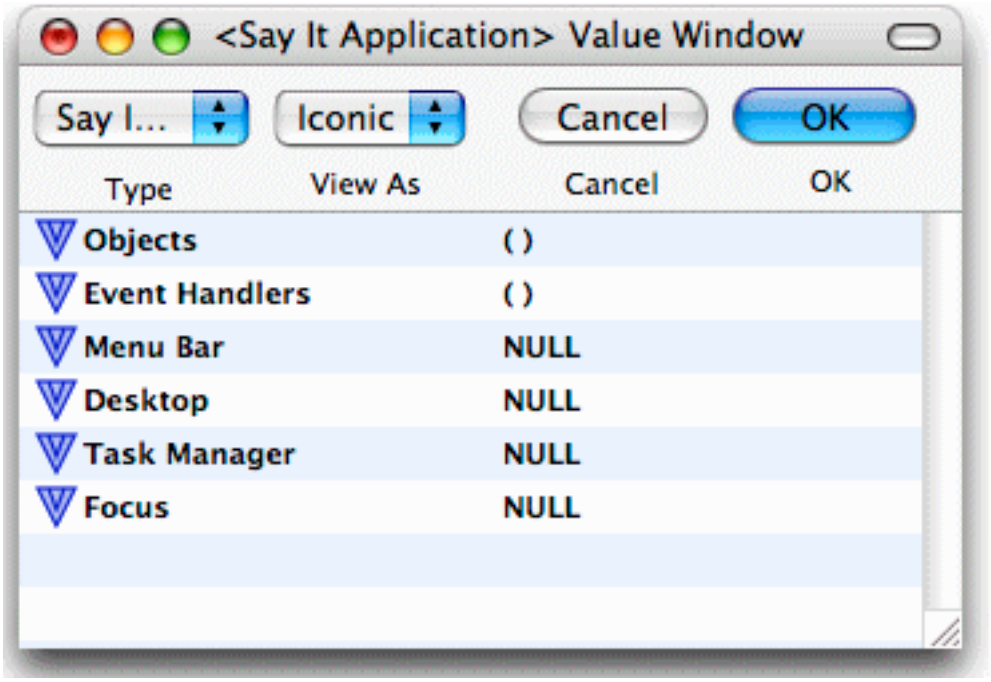
- Command-click in the Persistent window to create a new persistent.
- Name the persistent **The Application**.
- Double-click the new persistent item.

A NULL Value window opens.

The Value window is another of Marten's native editors. It lets you view,create, and modify simple data, lists, and objects. A key feature of the Value window is allowing you to specify or change types. In this case, you want to create an instance of the subclass you just created.

- From the **Type** popup, choose **Say It Application**.

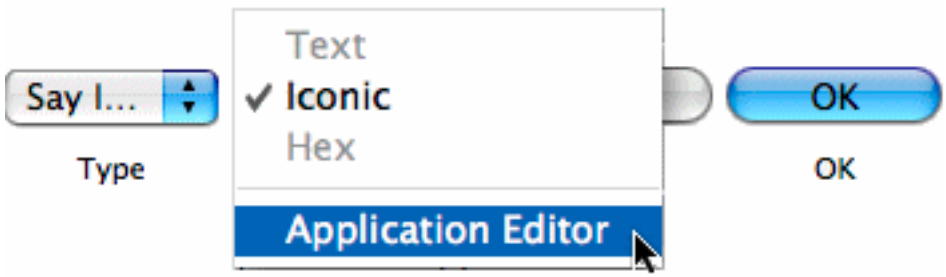
The title of the Value window is updated to indicate a **Say It Application** instance. Also, the Value window shows the default attributes that the **Say It Application** class inherited from **Small Application**.



Editors for working with class properties

While a Marten Value window provides the ability to edit the instance, the framework you loaded includes an editor that makes modifying the attributes of your Say It Application much more convenient.

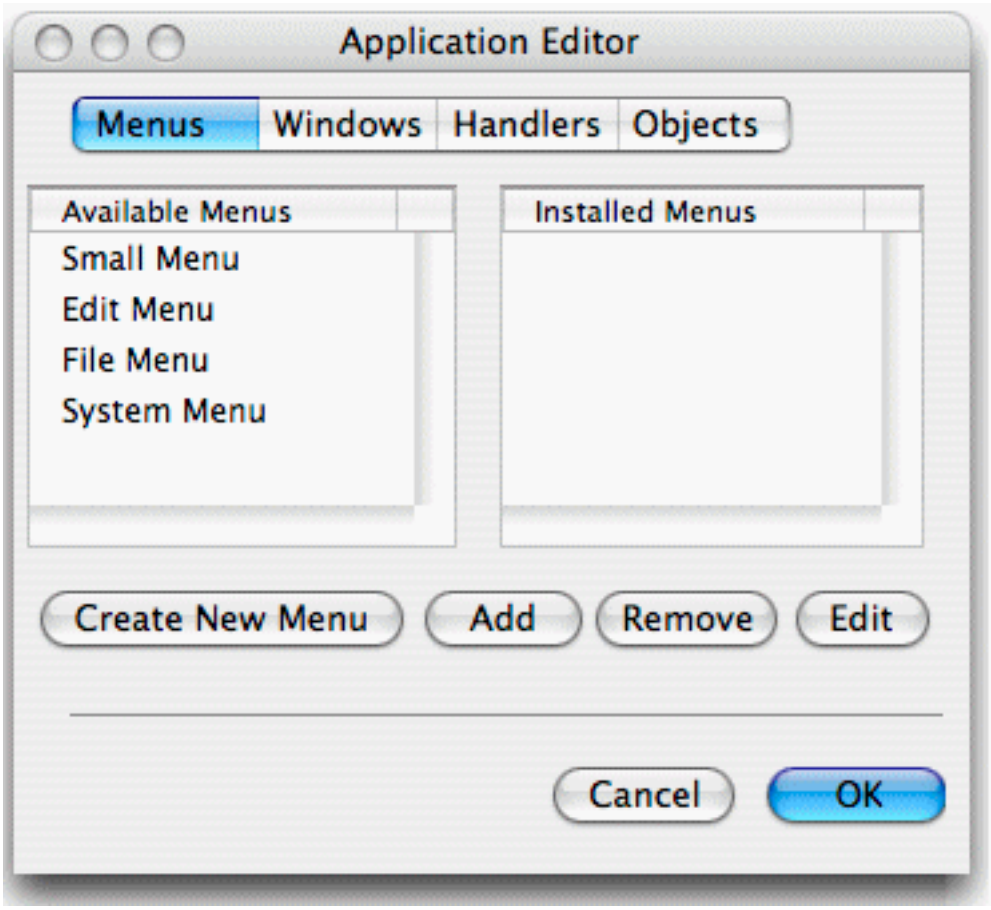
- From the **View** popup menu, and select **Application Editor**.



The editor for **Small Application** opens. The Application editor provides hierarchical, or drill-down, access to the various components of your application.At the highest level, you have access to the applications menus, windows, handlers, and objects.

Adding a menu

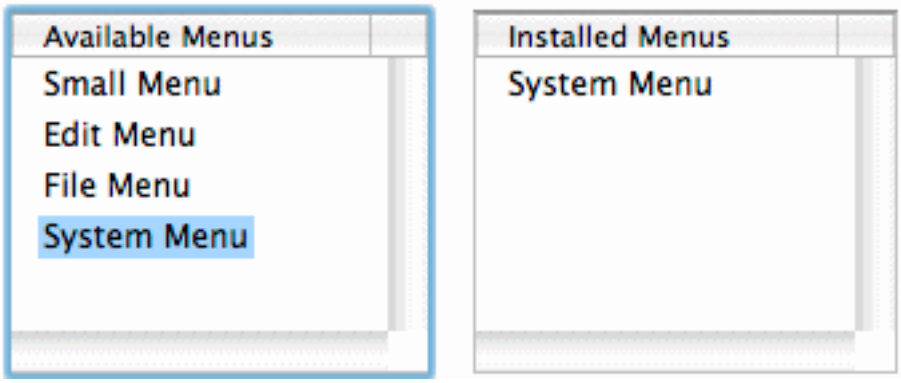
The initial display shows the available menus you can install.



A good debugging practice is to include the "canned" **System Menu** in your application. When an application is running in interpreted mode, the Marten editor cannot regain control of the application unless execution is suspended. This could cause a problem, for example, if a fault occurs within the application's event loop. The **System Menu** has a **Switch To Editor** command that will return control to the Marten editor.

- Select the **System Menu** item in the **Available Menus** list and click the **Add** button.

  A **System Menu** is added to the list of installed menus.



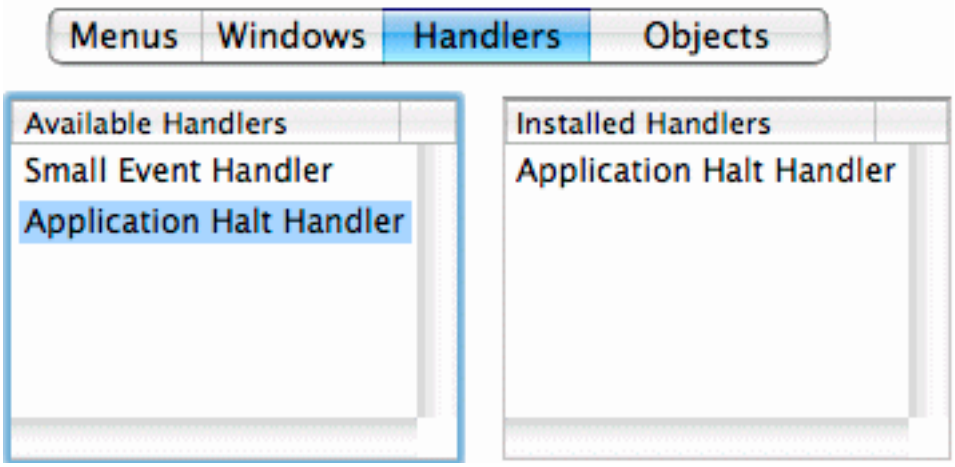
Adding an event handler

When you use the **Switch To Editor** menu command, a halt event is sent to the application. In order to process that halt event, you must install an event handler that designed for that purpose. There is a simple handler that is designed specifically to handle the halt event.

- Click on the **Handlers** segment of the top "segmented" view.

  The lists of available and installed handlers are shown.

- Select the **Application Halt Handler** and add it to the list of installed handlers.



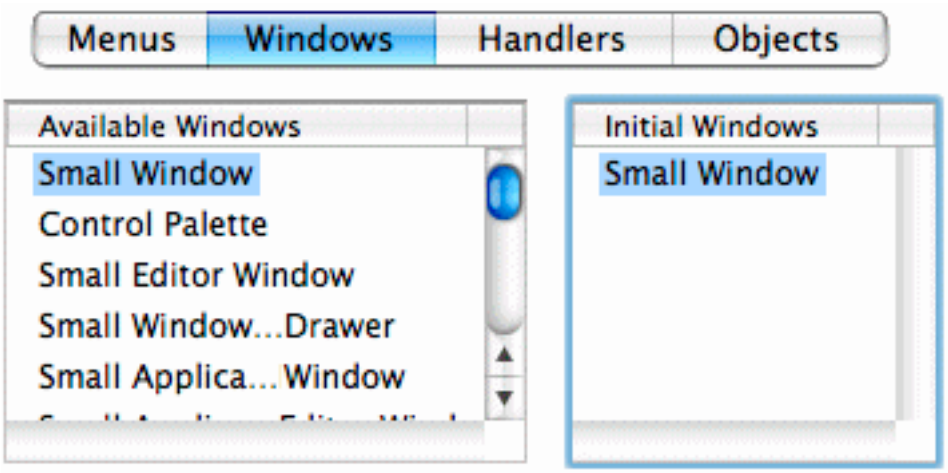
Adding a window

Next, you can set up your application's one window.

- In the Application editor, click on the **Windows** segment

  The lists of available and initial (installed) windows are displayed.

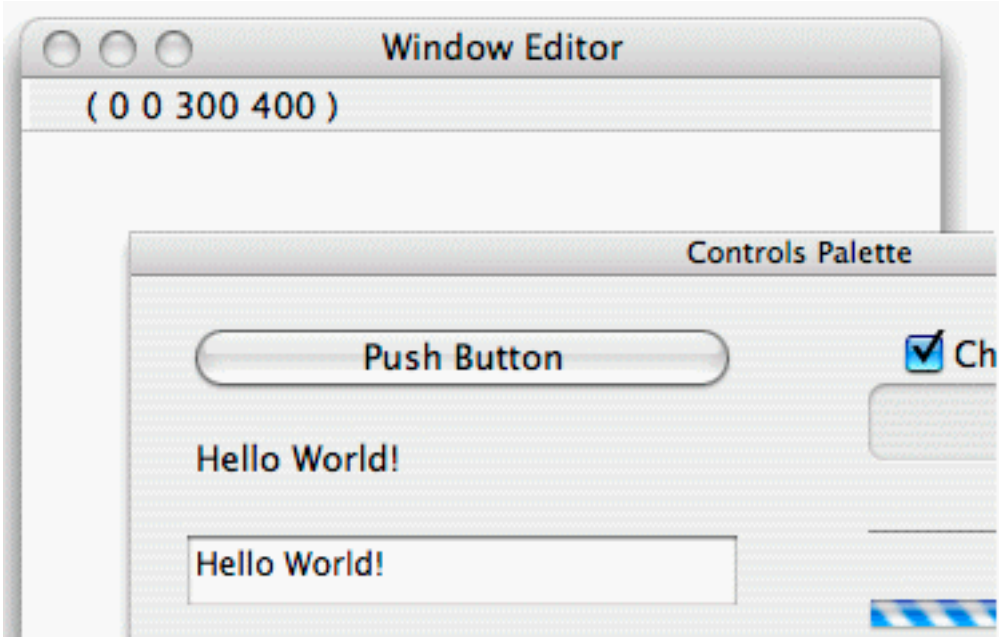
- Select **Small Window** and add it to the **Initial Windows** list.



- Select the **Small Window** instance in the **Initial Windows** list and click the **Edit** button to open a Window editor.

  A window editor opens.

This editor is a bit more complex and initially consists of two windows, a WYSIWYG Window Editor and a palette of controls that can be dragged and dropped onto the Window Editor.



The **(0 0 300 400)** designation shows the current frame. If nothing is selected, it is the frame of the edited window content region which is initially 300 vertical pixels by 400 horizontal pixels.

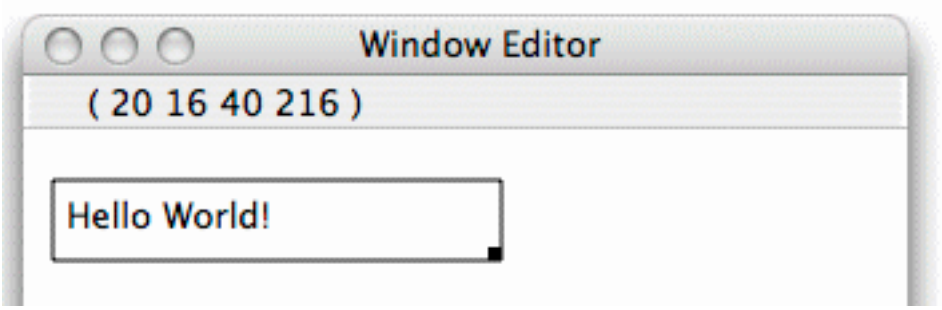
The Say It window will have three controls:

- A static text control
- An edit text control
- A push button control.

You can start by adding the static text control to the editor.

➤ Mouse down on a static text control and drag it onto the editor.

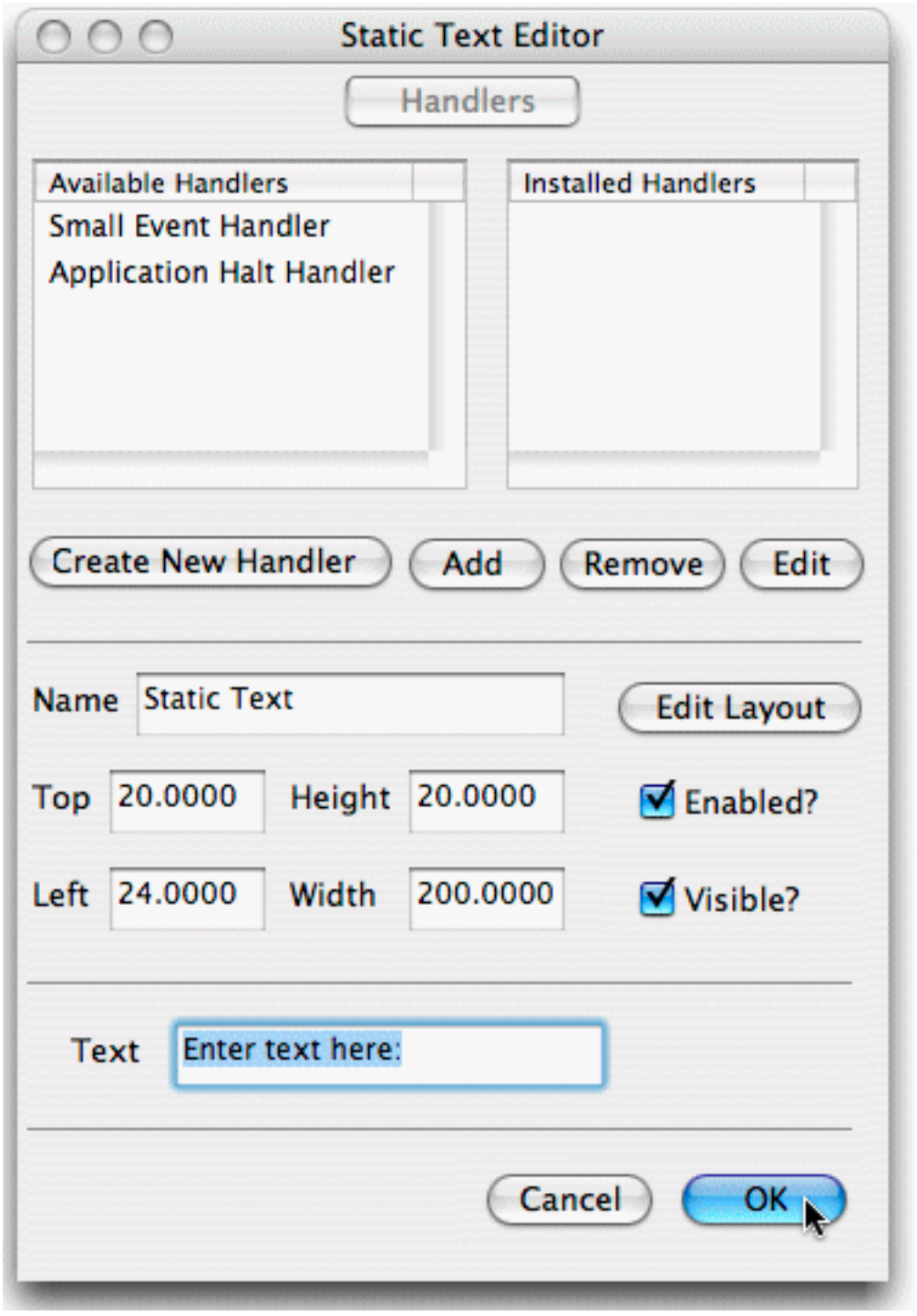
A new static text control appears in the editor.



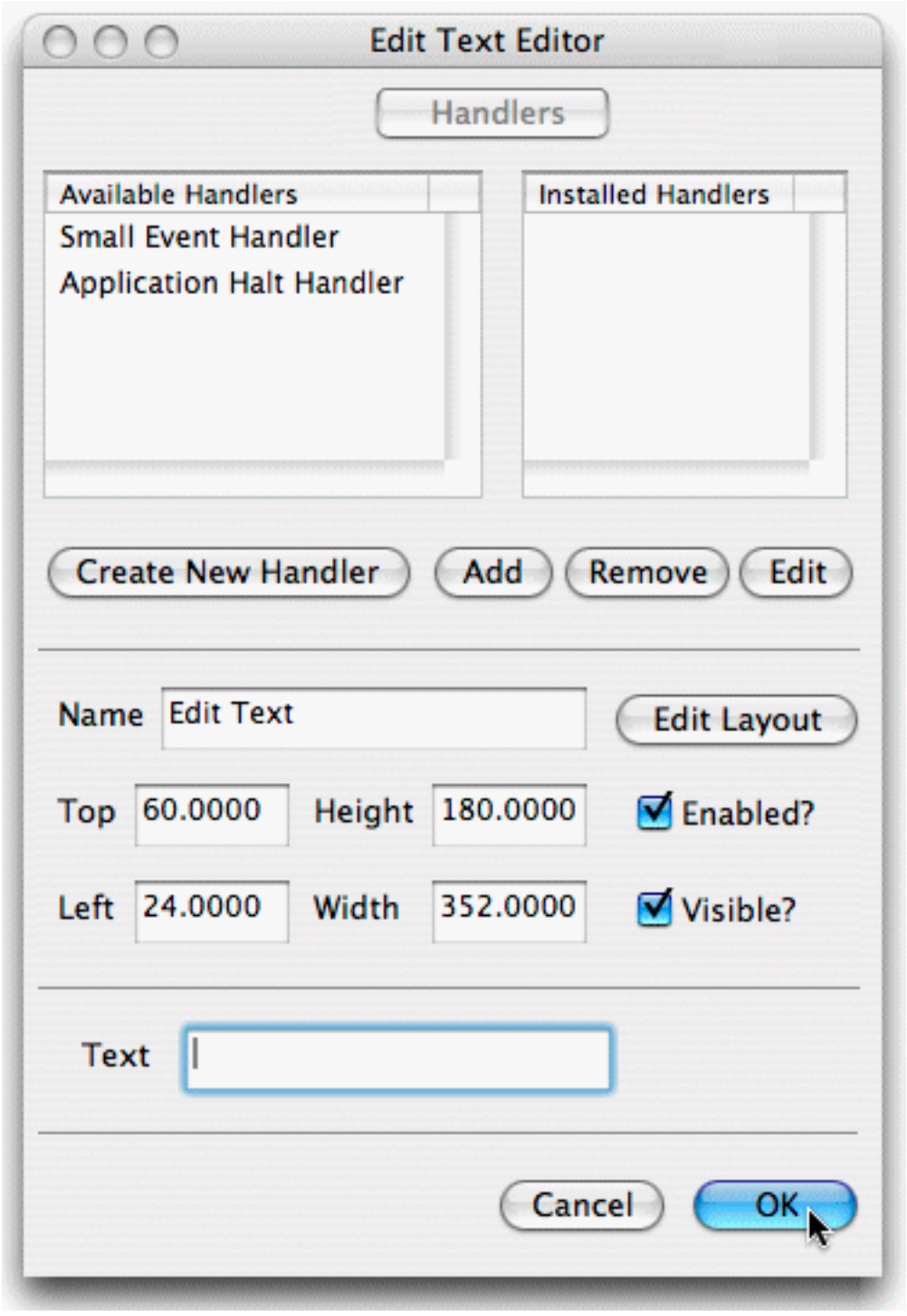
This editor lets you drag controls to position them but you can position things more accurately by opening another editor.

➤ Double-click directly on the new control.

A Static Text editor opens.



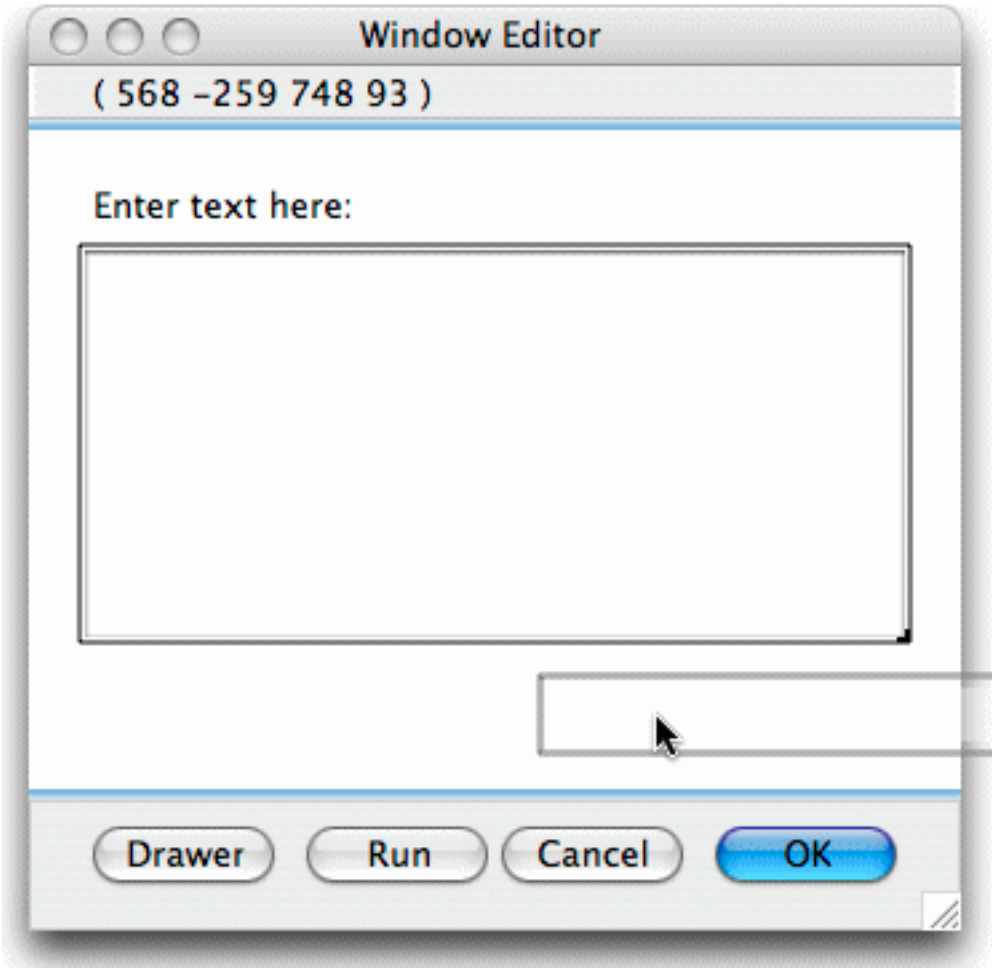
- Change the dimensions and text to match the above and click the **OK** button.
- Double-click the edit text control to open its editor.



- Change the dimensions and values to match those above and click the **OK** button.

Adding a push button to the window

The Window Editor should now look something like the following.

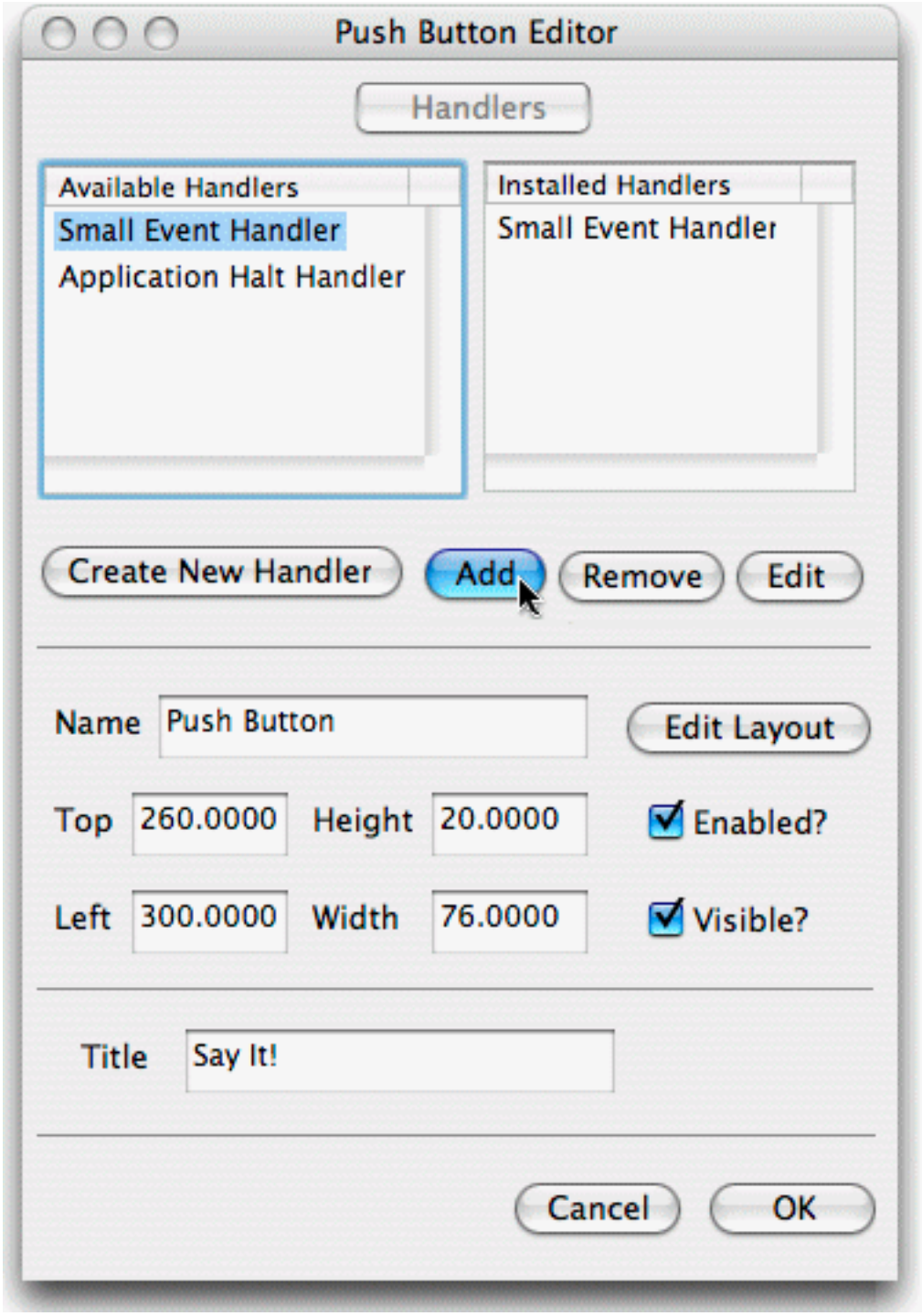


- Drag the push button control onto the canvas
- Double-click the new push button.

  A Push Button editor opens.
- Set the title of the button to **Say It!** and edit the attributes of the push button to those shown below.

Now unlike the other controls, you want this control to take some action when the push button is clicked. When the button is clicked, a Control Hit event is sent to the button. Consequently you will have to install an event handler so that the button can handle this click event.

- Select a **Small Event Handler** and add it to the installed list of handlers.



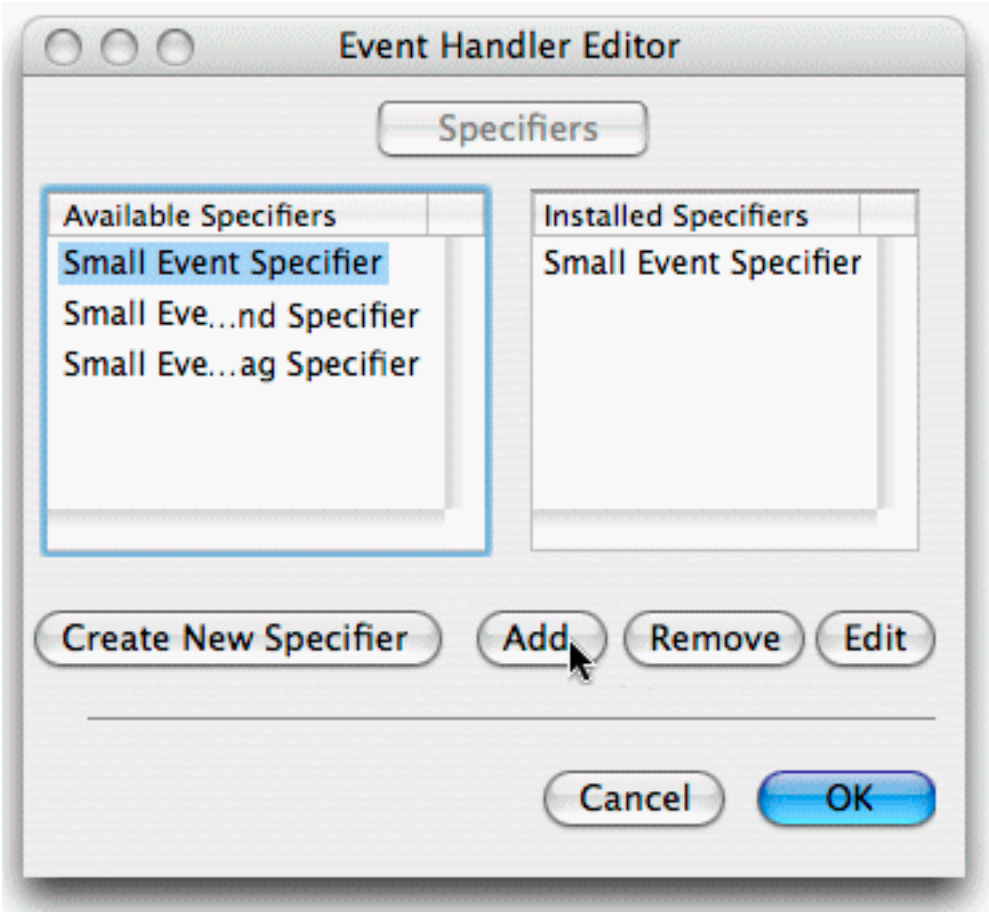
The Application Halt Handler that you added earlier was specific in the events it handled. The Small Event Handler is not. It is just a generic handler of events and you have to specify which events to handle. To do this, you edit it.

- Select the **Small Event Handler** in the list of installed handlers and click the **Edit** button.

  An Event Handler editor opens.

For such a simply functioning control, you will only need to handle a single event. The event you are interested in handling is a Control Hit event.

- Select a **Small Event Specifier** and add it to the installed list.

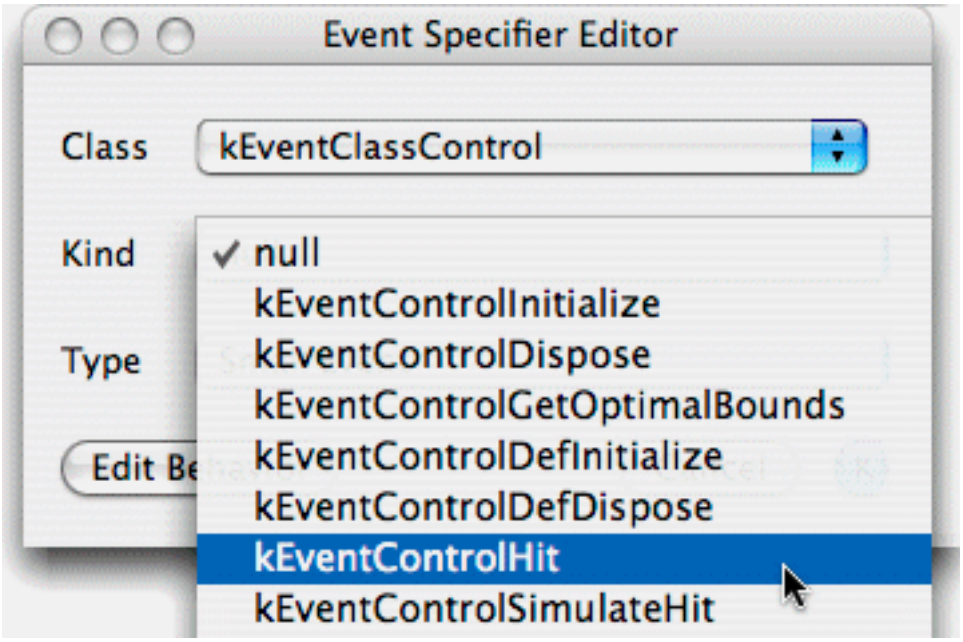


Next, you must specify the event to handle, a Control Hit event.

- Select **Small Event Specifier** in the installed list and click the **Edit** button.

  The Event Specifier editor opens.

- From the **Class** dropdown, choose **kEventClassControl**.
- From the **Kind** dropdown, choose **kEventControlHit**.



Now you can specify the behavior that responds to the event. This is the method that will execute.

- Click the **Edit Behavior** button.

  A Behavior editor opens.

You want to invoke the now-familiar **Speak-Text** method. Shortly, you will make a minor modification to the method so that it works in this context.

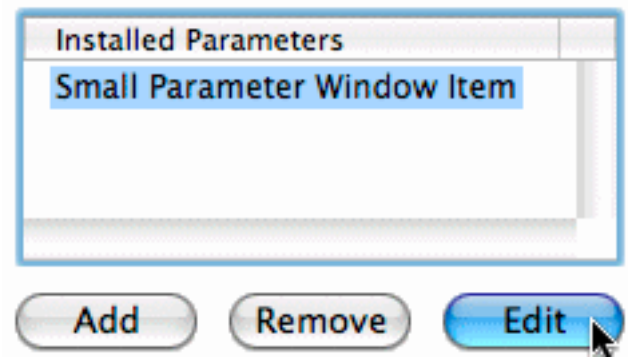
- Set the **Method Name** to **Speak-Text**.

You want the application to "speak" the text currently populating an edit text control. Now you should specify that window item as an input or "parameter" for the behavior.

- Select a **Small Parameter Window Item** and add it to the installed parameters list.

You will have to edit the window item to make the Edit Text control the input.

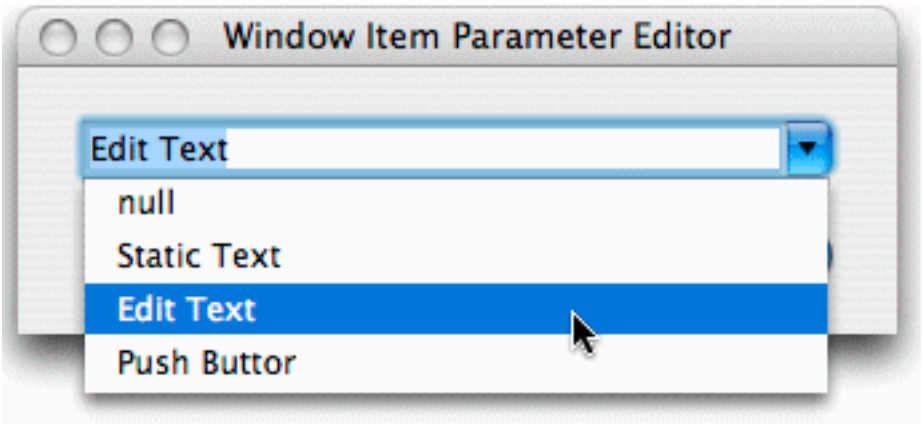
- With the **Small Parameter Window Item** selected, click the **Edit** button.



A Window Item Parameter Editor opens.

It has a combo box that discloses a list of the names of the available controls in the window. Specifically, you want the edit text control named **Edit Text**.

- Select **Edit Text** from the combo box and click the **OK** button.

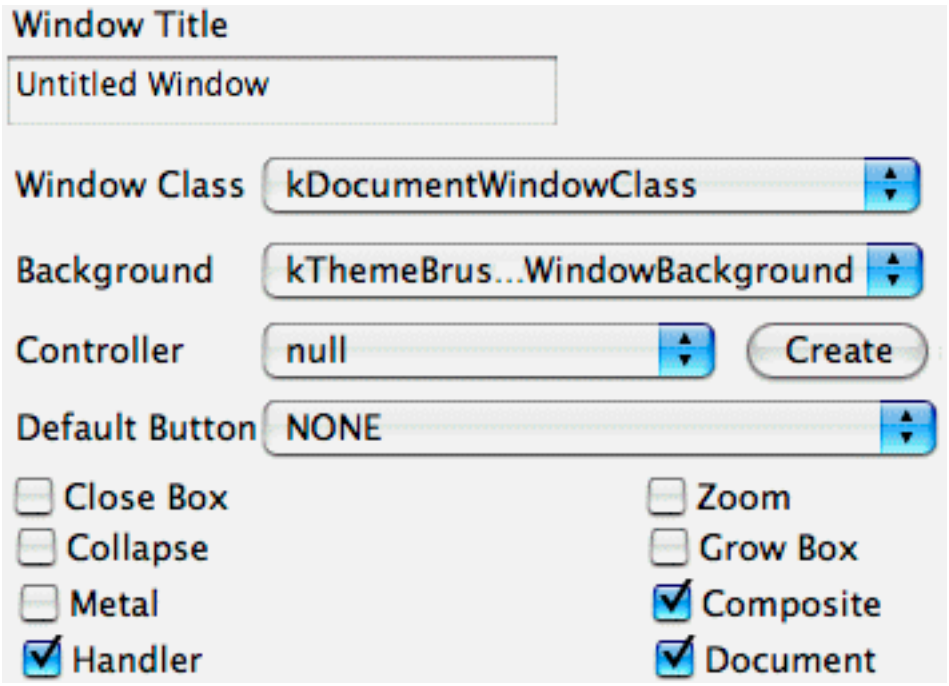


- Continue to close the editors by clicking the **OK** buttons until you are back at the Window Editor.

Now you need to edit some properties of the window, like its name and background. These properties are accessed by opening a drawer.

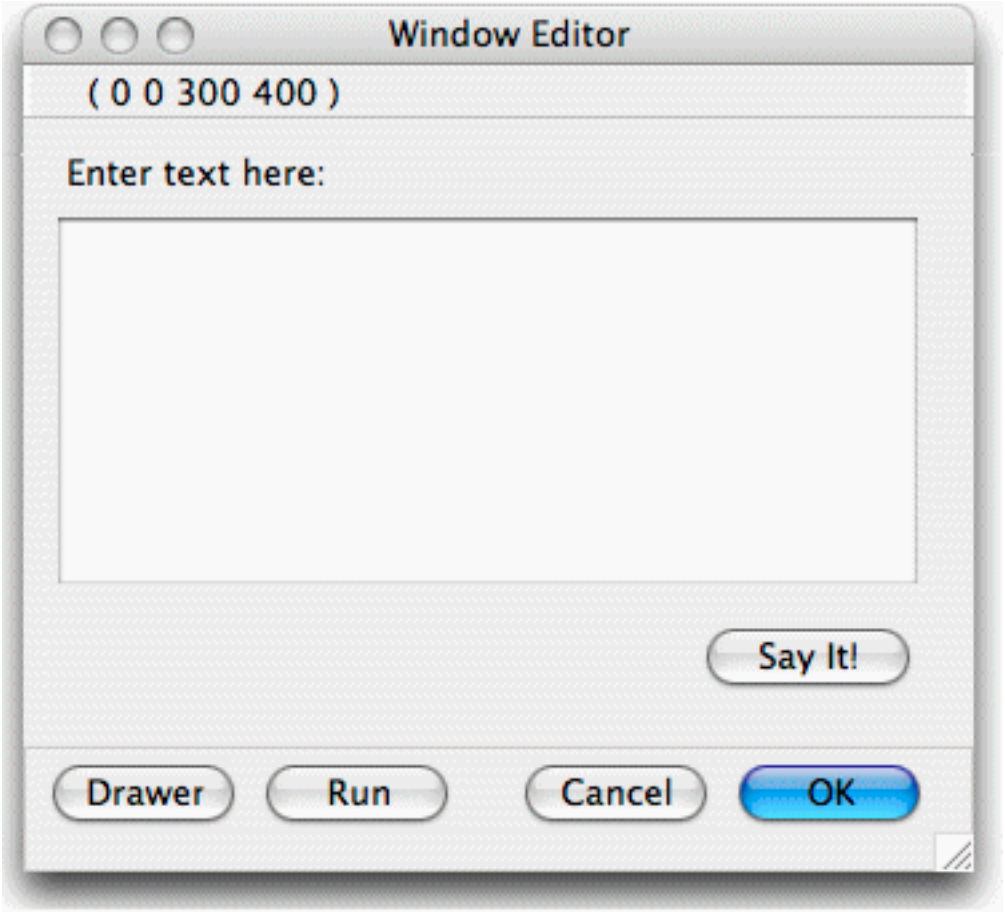
- Click the **Drawer** button.

The properties drawer opens.



- Change the title to **Say It!**.
- Choose **kThemeBrushUtilityWindowBackgroundActive** from the **Background** dropdown.
- Click the **Drawer** button again to close the drawer.

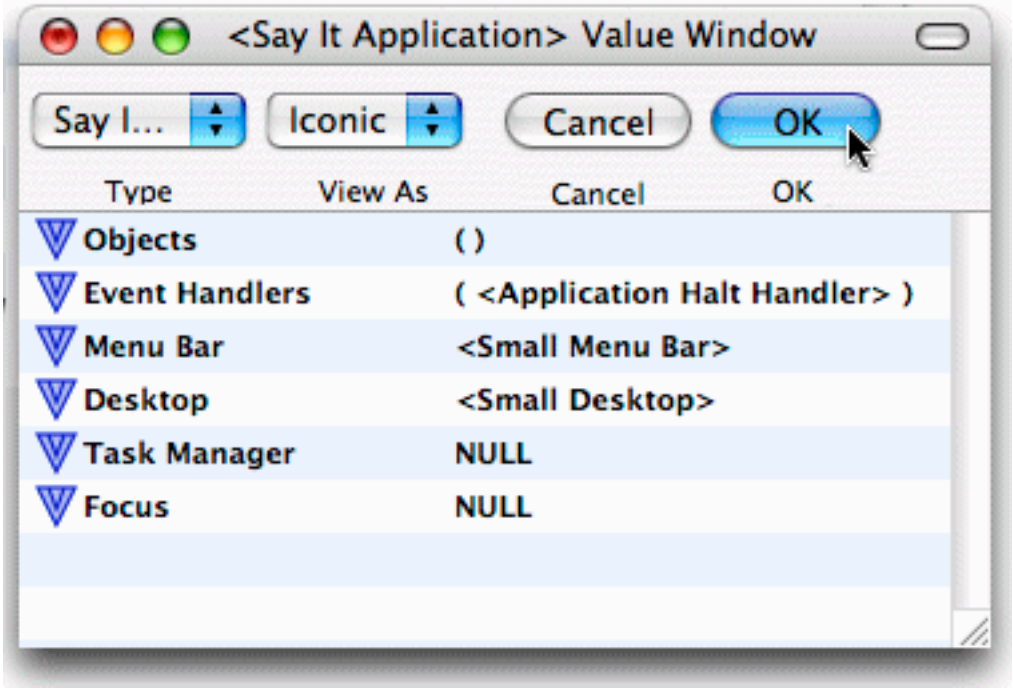
The window should now look like the following:



This completes your window editing.

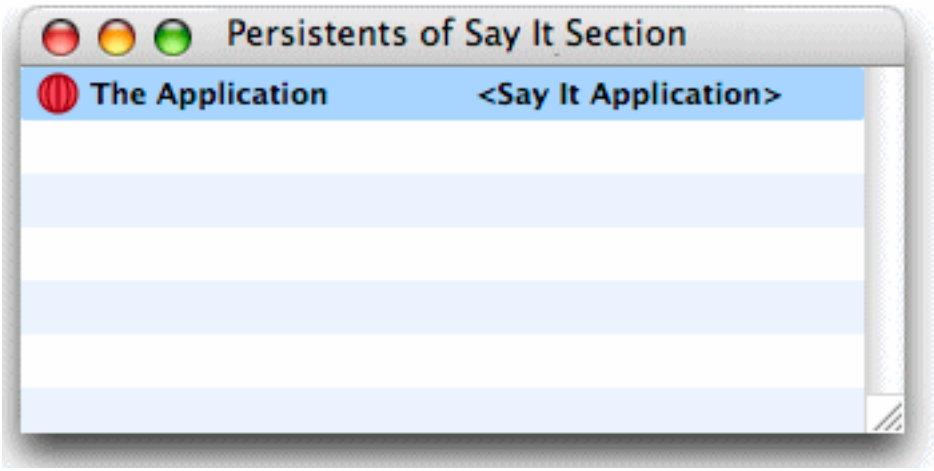
- Click the **OK** button on the window editor.
- Click the **OK** button on the Application editor.

This returns you to the original **Say It Application** Value window. It now displays attributes with your newly edited values.



- Commit these changes by clicking on the **OK** button.

The Application persistent now displays a value of an instance of the **Say It Application** class.

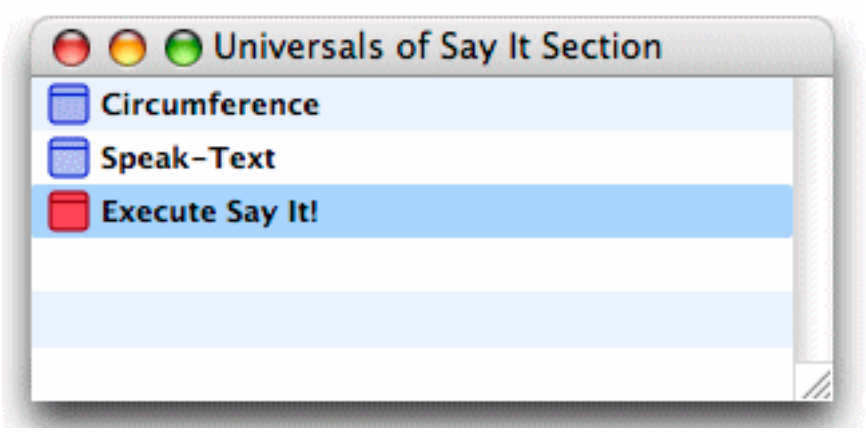


Specifying a MAIN method

When a user double-clicks on your final application, the application has an initial method that executes. In Marten, this is referred to as the MAIN method. In development, you must designate a particular method to be the "MAIN" method. This

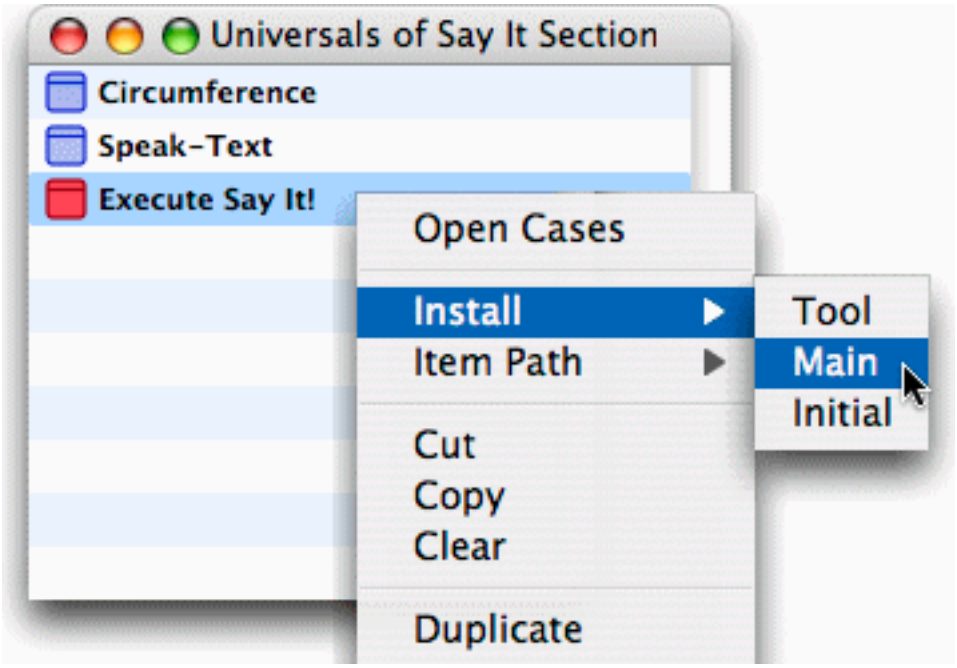
method can do anything but is most useful when it executes an instance of an application. You will now create such a method to execute your Say It application.

- Open the Universals window for the **Say It** section.
- Create a new universal named **Execute Say It**.

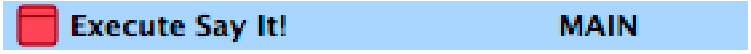


Marten makes it easy to designate the MAIN method.

- While holding down the CONTROL key, click on the **Execute Say It!** method, choose **Install** from the context menu, and then choose **Main**.



Display of the method in the Universal Methods window changes to indicate that the universal item is now the MAIN method for the project.



Key universal method updates

To account for the changes you've made, you'll have to make the following modifications:

- Have your **Speak-Text** method accept input from the edit text in the window you created
- Have your MAIN method execute the application instance.

Modifying your Speak-Text method

Earlier, in "Adding a window" on page 43, two of the keys tasks were:

- Adding an edit text where users can type the text to be spoken
- Adding a push button to initiate speaking the text.

When defining the push button's behavior, you specified that upon receiving a Control Hit event, the application pass the current instance of the edit text as an input to a universal called **Speak-Text**.

Currently, your **Speak-Text** method prompts the user for text using the **ask** primitive. You'll have to modify it to accept an instance of the edit text as an argument to the method.

- Open the **Speak-Text** universal method.
- Click on the **ask** operation and press DELETE.
- In the same location, COMMAND-click to create a new operation.
- Name it **/Get Value**.

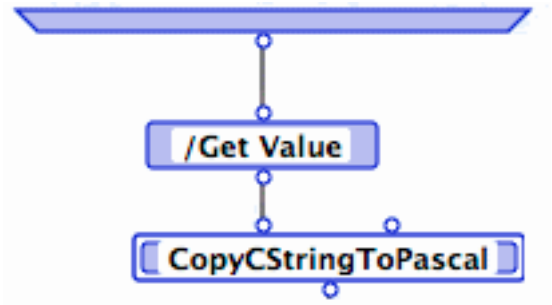
This is a call to a class method. Marten provides a number of different takes on calls to methods. In this case, the slash preceding the method name, specifies a data-driven reference. Since the input to the method will be an edit text instance, Marten will execute a method named **Get Value** defined for that class.

- Click on the terminal at the top of the **/Get Value** operation.
- With that terminal selected, OPTION-COMMAND-click very close to the bottom of the input bar.

This is a shortcut for creating a datalink to another operation and simultaneously creating a root on the other operation. The new root on the input bar is the method by which parameters are passed into the method.

- Click on the root at the bottom of the **/Get Value** operation.
- With that root selected, OPTION-click on the rightmost terminal of the **CopyCStringToPascal** operation.

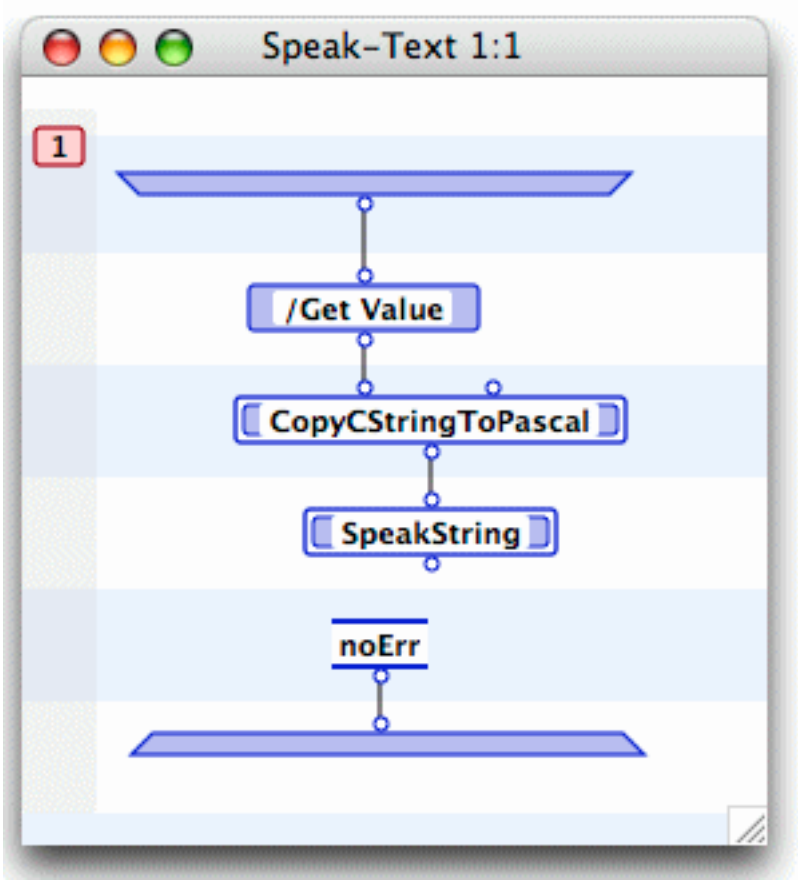
By now, the affected area of your case should look like the following:



Lastly, an event behavior must return a constant indicating whether the event was handled successfully or not.

- COMMAND-click just above the top of the output bar to create a terminal.
- Double-click the new terminal to create a new constant operation with a single root, connected by a datalink to the input bar.
- Name the constant **noErr**.

You method should now look like the following:

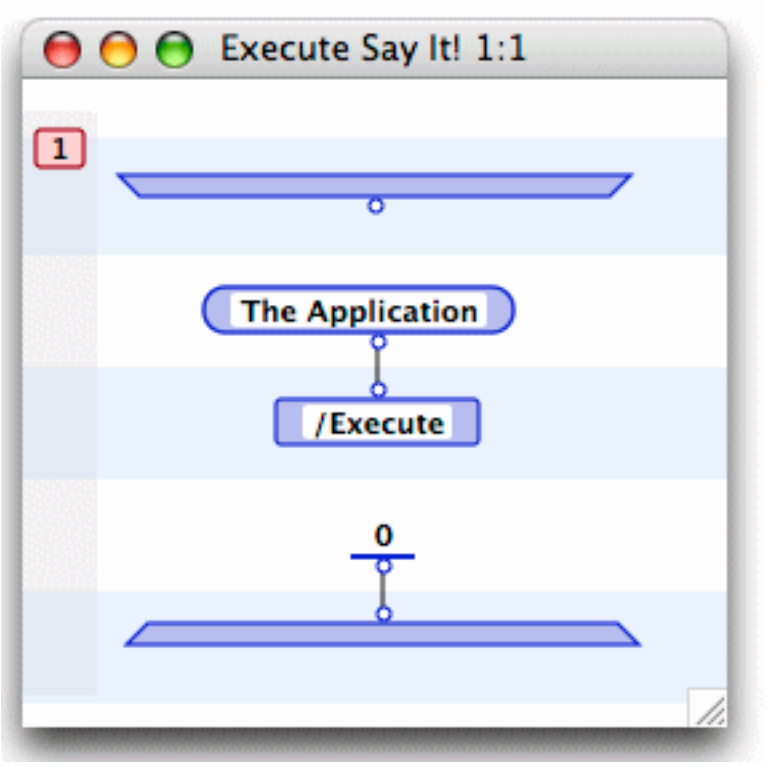


Modifying the Execute Say It! MAIN method

As well, you have to modify the **Execute Say It!** universal to actually run the application you've created. Remember: all of the application attributes you worked with in previous steps are stored in a persistent called **The Application**. The **Say It Application** class has a method called **Execute** that it inherited from the **Small Application** class. It will generate your application from the properties you provided in the persistent, **The Application**.

In addition, a MAIN method has a list of command line argument strings as input (which we ignore) and must return a constant to the system. The standard value for this constant is **0**, so the MAIN method should return **0**.

- Open the **Execute Say It!** method and edit it so that it looks like the following:



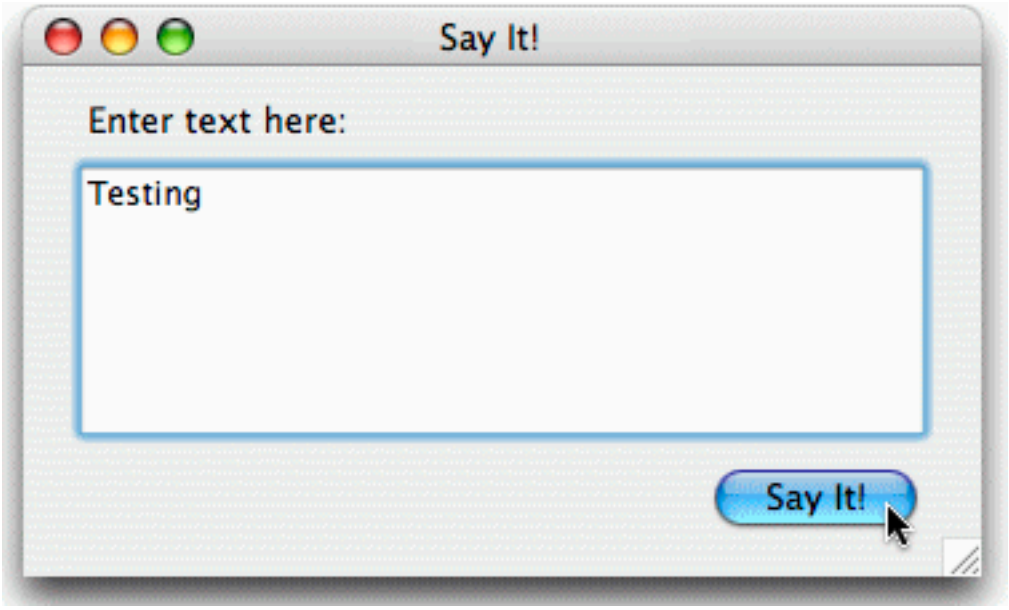
- Close the method.

Running your application

Previously, in order to inspect execution, you ran an individual method. Now that your application has a MAIN method, the **Run Application** command is available.

- From the **Run** menu, choose **Run Application**.

The application runs and puts up a **Say It!** window.



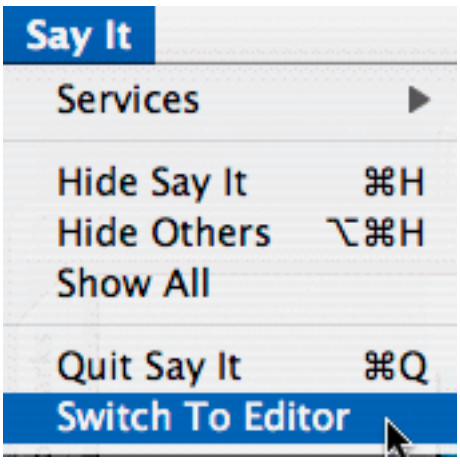
- Type **Testing** into the edit text box and click the **Say It!** button.

  The text is spoken.

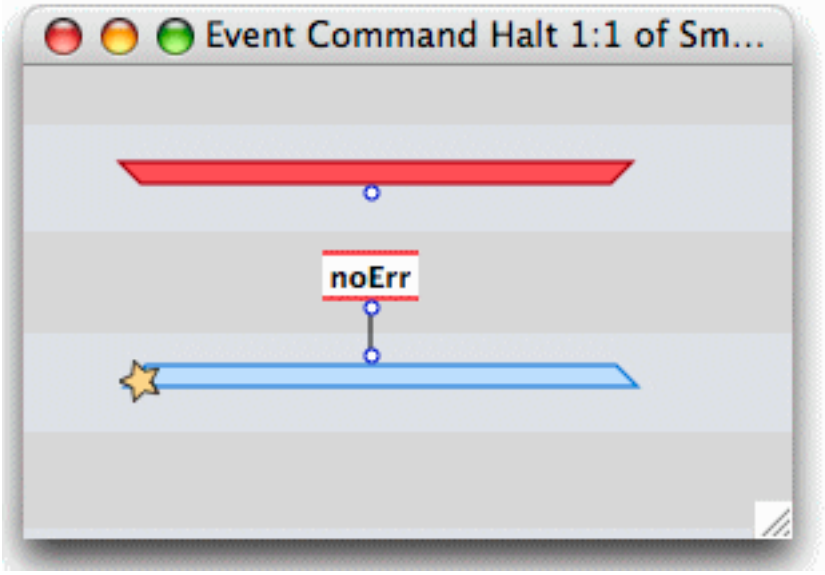
Modifying code on the fly

After the text has been spoken it remains in the edit text box. One change you could make is to clear the text from the edit text box each time the text has been spoken. Marten has the ability to edit code while a method is executing if the method can be halted. You accounted for this eventuality by installing the standard System menu and the Application Halt handler. Consequently, a new menu item has appeared under the **Quit** menu command in the Say It system menu.

- From the System menu, select **Switch To Editor**.

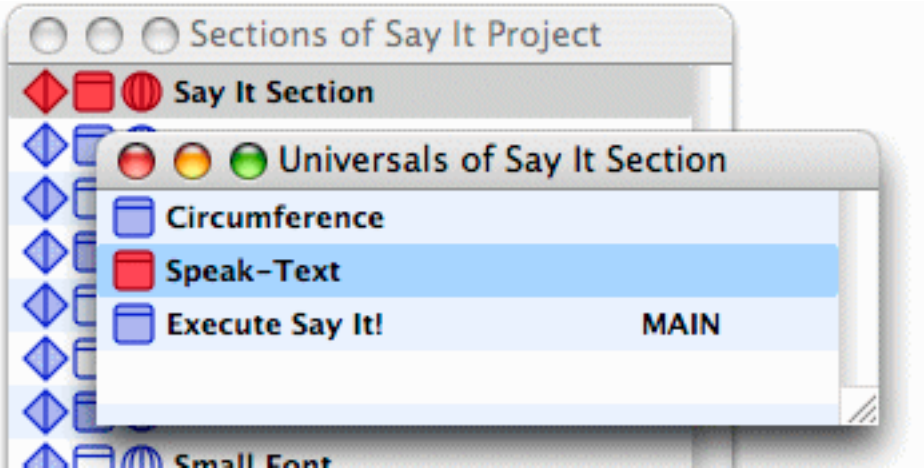


The halt mechanism is a simple breakpoint, so the method halts and a debug window opens up at the breakpoint. At this point, control has been returned to the Marten IDE process and you can now edit a case, add a universal, delete a class. All normal functionality of the Marten IDE is available EXCEPT the ability to run another, different method. If you try, this Event Command Halt method will resume instead. Currently, only one method in a project can be executing at any one time.

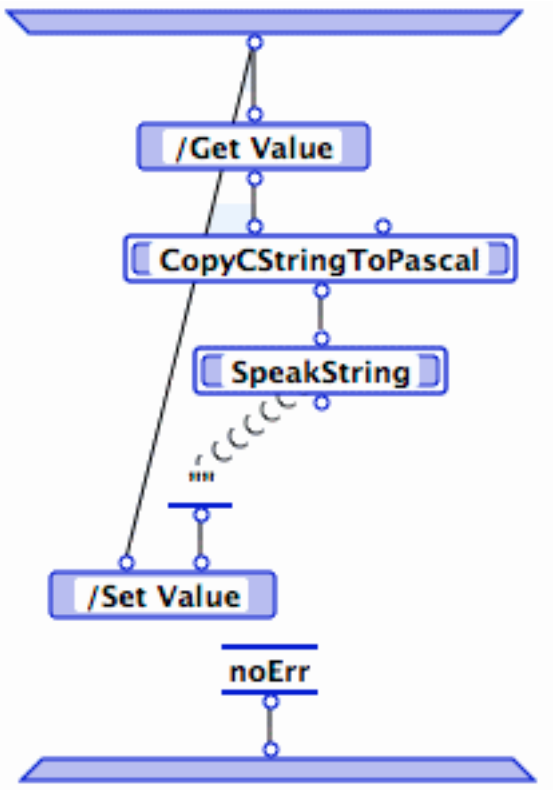


Once the application has been halted by hitting the breakpoint in the "halt" method, you are free to make any changes you want to the code. In particular, can now modify your **Speak-Text** method to clear the text from the edit text box after the text has been spoken.

- Open the **Speak-Text** method.



- Modify it to look like the following:



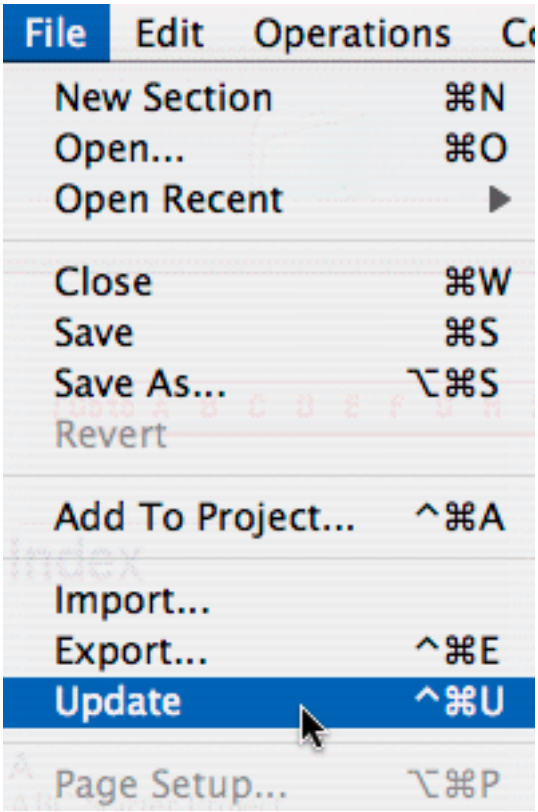
Notes:

1. If you've forgotten how to create a synchro, see "Controlling sequence of execution" on page 26.
2. The constant passing the value to the rightmost **/Set Value** terminal is an empty string.

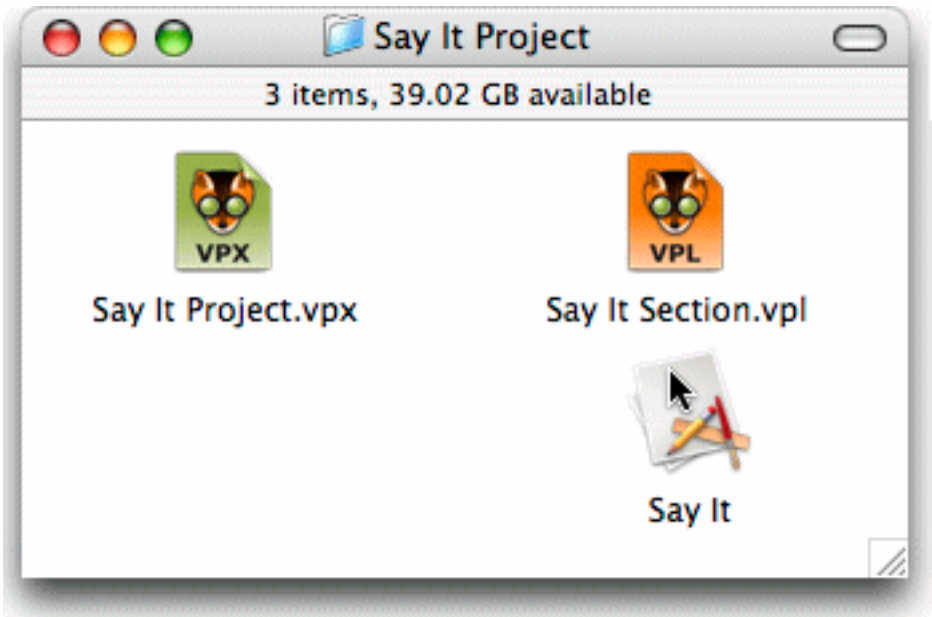
Generating a standalone application

Now that you have your application working the way you want it to, you can easily generate a standalone application, one that sits on the desktop and can be double-clicked to launch. To create your standalone application, you need to update your project application with all of the code you just created.

➤ From the **File** menu, choose **Update**.



Your project application is now a standalone application.



Double-click it. It runs just like it did in the Marten environment.

7

Summary

This tutorial has walked you through the basics of Marten programming and touched on many of the high points of the Marten development environment. The following topics tell you where to go next to continue your progress in becoming an expert in the use of Marten:

Base documentation set

In addition to this guide, the base documentation set for Marten has the following titles:

Marten User Guide	This book is the definitive guide to use of the Marten development environment. It provides details on the Marten language, basic operations, project and section maintenance, editing and debugging features, and direction on creating compiled applications.
Marten Primitives Reference	This book provides everything you need to know to work with the primitives packaged with Marten. This includes details on the contents of each library containing primitives, and how to locate and load libraries, as well as detailed description of the purpose, inputs, and outputs of each primitive packaged with Marten.

Sample code

Complete Marten source code for a number of sample applications is packaged with Marten. By default, it is installed in the **/Samples/Marten** folder. Inspecting this code and watching it execute is valuable experience in becoming an experienced Marten programmer.

Other sources of information

The Support page on the Andescotia Web site is a source of additional documentation, examples, and other helpful resources. It is available at http://www.andescotia.com/support/.

Index

H

I

L

M

N

O

P

R

S

T

U

V

W


A B C D E F G H I J K L M N O P Q R S T U V W X Y Z