Simplest Editor Tutorial

This tutorial illustrates how to create a very simple text editor using just the standard primitives that are supplied with the Marten IDE. In particular, you will learn how to use the basic file manipulation primitives to open a file, read the contents and close the file. In addition, you will also learn how to display the text you retrieved from the file in a modal dialog window (modal means that the display captures all user interaction).

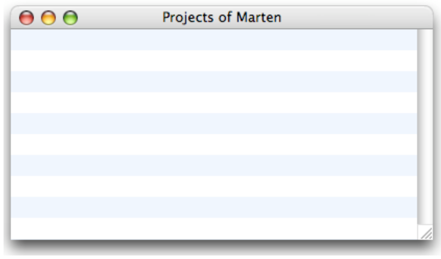
Set up

To follow this tutorial as written, create a folder named "Simplest Text Editor. This folder is where you will store all the files related to the project.

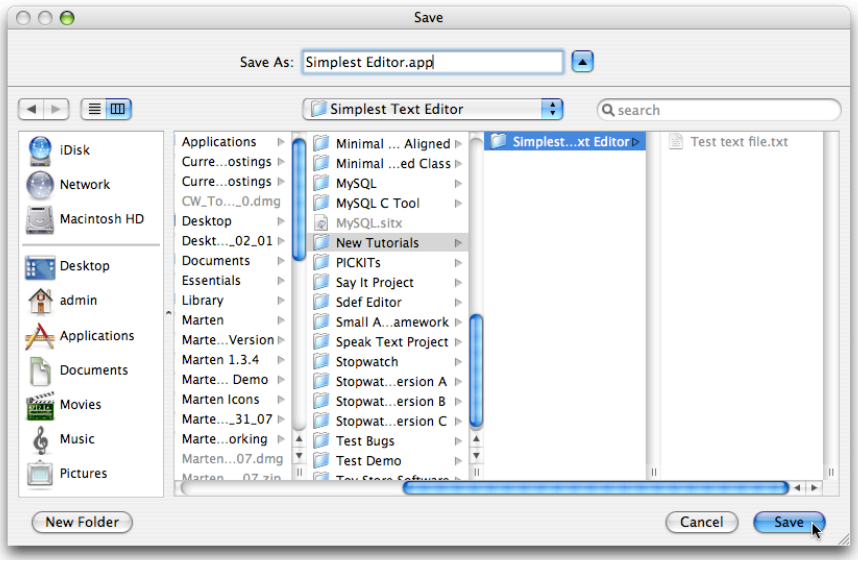
In addition, you will need a plain text file for use in testing the text editor. This file should be named "Test text file.txt" and can be created by the TextEdit application that ships with MacOS X. To create this file, launch TextEdit and type "My text file!" into the document window. Then select the "Make Plain Text" menu item from under the Format menu. This will convert the file to plain text. Finally, save the document in the project folder and name it "Test text file.txt".

Creating the project

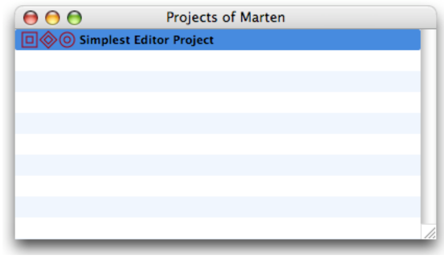
With both the project folder and the test file created, you are ready to start coding. Launch the Marten application. The "Projects of Marten" window opens.



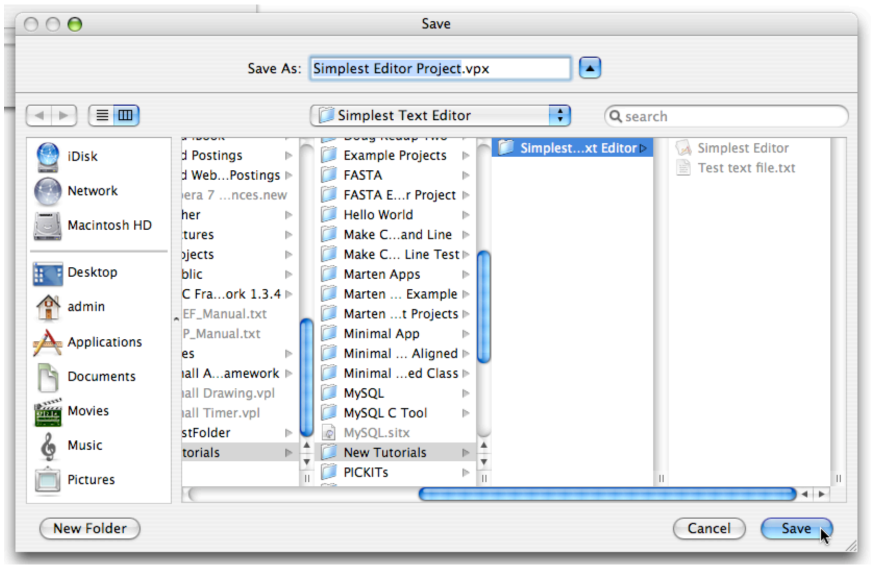
Next, create a new project by command-clicking in the window. A Save dialog will open asking where to save the project application. Name the project application "Simplest Editor.app" and save it in the project folder.


Save As: Simplest Editor.app

A new project item appears in the Projects window. Rename the item to "Simplest Editor Project".



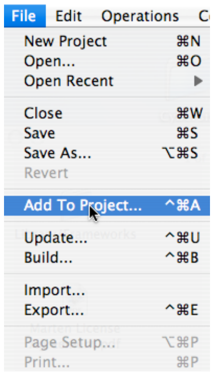
Save the project by either selecting "Save" from the File menu or typing Command-S. A Save dialog will open. Accept the suggested file name ("Simplest Editor Project.vpx") and save the file in the project folder.



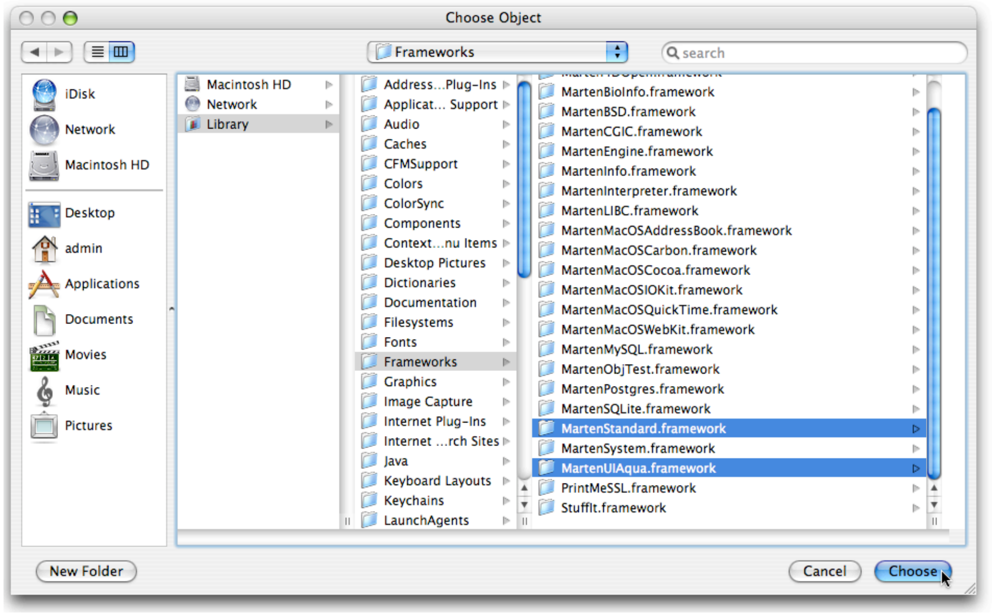
Adding the Marten libraries

The Prograph language specification does not implement such operations as "+" or "bit-and". It does allow for such "primitive" operations to be supplied externally in libraries. Marten comes with several libraries which provide for standard operations such as multiplication and string length. These Marten libraries come in the MacOS X "XXX.framework" folder format and as such, should generally be installed in the root /Library/Frameworks/ directory. For the Simplest Editor application project, you need to add the two basic Marten libraries to the project. These libraries are the MartenStandard library which provides standard platform-independent primitives and the MartenUIAqua library which provides primitives that manipulate MacOS X specific structures such as rectangles and importantly for this project, file specifications.

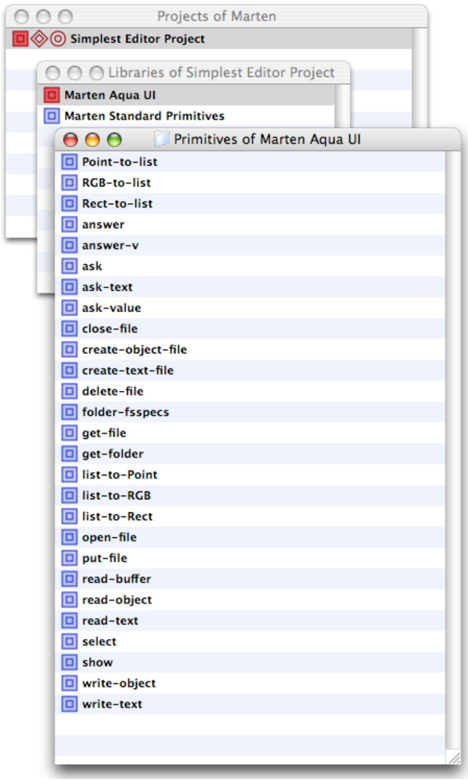
To add these libraries, select the "Add To Project..." menu item under the File menu.



A "Choose Object" dialog will open. Navigate to the /Library/Frameworks/ folder and select both the "MartenStandard.framework" folder and the "MartenUIAqua.framework" folder.



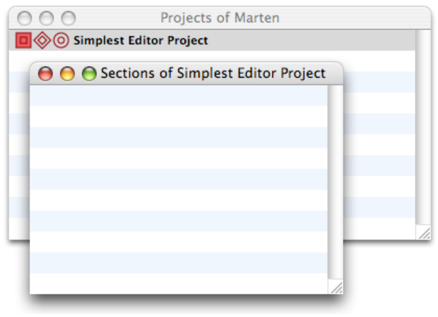
To see if the libraries were successfully added to the project, double-click the libraries icon of the project item (the square icon to the left of the diamond and circle). A "Libraries of Simplest Editor Project" window will open with two library items displayed. If you double-click the primitives icon, a list of that library's primitives is shown. For the Marten Aqua UI library, the primitives include the get-file, open-file, close-file and other file manipulation primitives which we will use to get the data for our editor. The primitives also include GUI text display primitives such as ask, answer, and show, which we will use to create the "window" of our editor.


Projects of Marten
Simplest Editor Project
Libraries of Simplest Editor Project
Marten Aqua UI
Marten Standard Primitives
Primitives of Marten Aqua UI

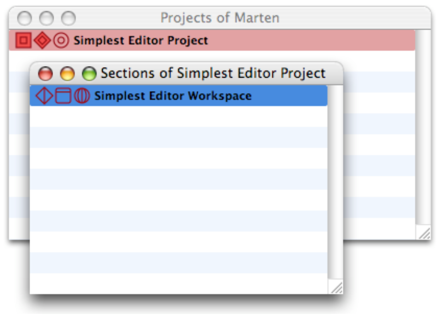
Point-to-list
RGB-to-list
Rect-to-list
answer
answer-v
ask
ask-text
ask-value
close-file
create-object-file
create-text-file
delete-file
folder-fsspecs
get-file
get-folder
list-to-Point
list-to-RGB
list-to-Rect
open-file
put-file
read-buffer
read-object
read-text
select
show
write-object
write-text

Creating a section to store the code

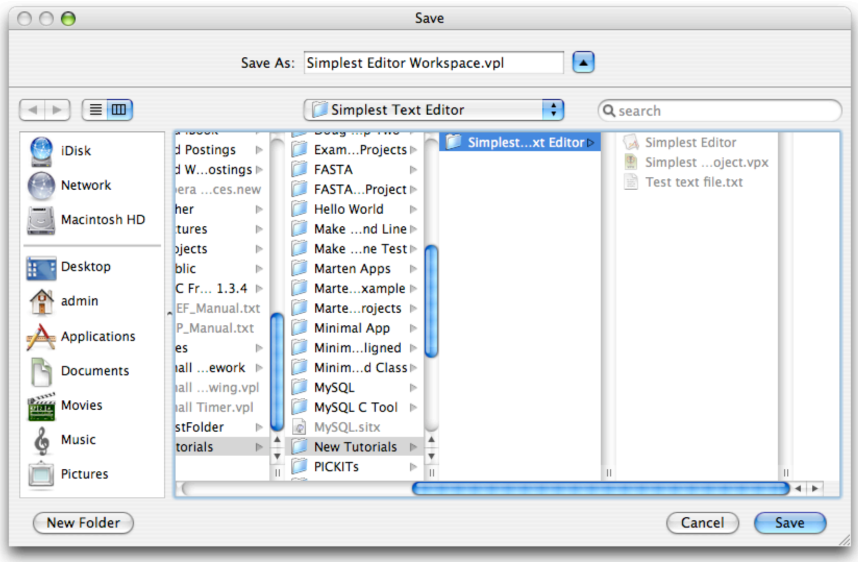
A project simply organizes the various pieces that make up an application. These include the libraries, resources such as pictures and icons, and of course, the code itself. Code is stored in section files and we need at least one. Add a section by double-clicking the sections icon of the project item (the middle "diamond" icon) to open a "Sections of Simplest Editor Project" window.



Command-click in the window to create a new section and rename it "Simplest Editor Workspace".


Projects of Marten
Simplest Editor Project
Sections of Simplest Editor Project
Simplest Editor Workspace

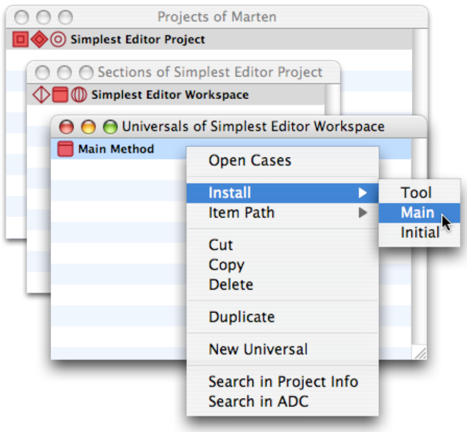
Save the section and a Save dialog opens. Accept the suggested name and save the section file in the project folder.



Creating a method

The Prograph language stores code in "cases" which in turn are stored in "methods". Methods are the named routines or functions of the Prograph language. Create a method by double-clicking the methods icon of the section item (the middle "rounded square" icon). Command-click in the window to create a method item and rename it "Main Method".

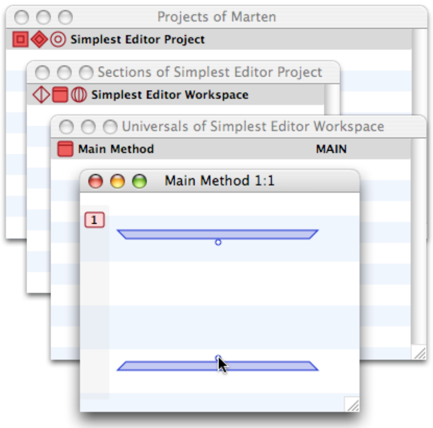
Now every application needs an entry point into the code. In the C programming language, this is a special routine named "main". The Prograph language does not specify an entry point so the Marten IDE annotates a method to be the entry point or "main" method. To make the "Main Method" the entry point, mouse-down on the Main Method method item while holding the Control key down. A contextual menu will appear. Navigate to the Install->Main menu item and select it.



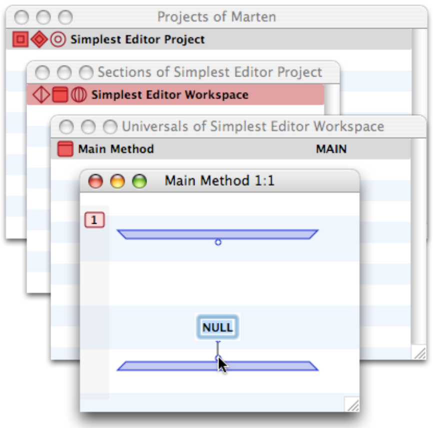
Creating the code

As stated before, code is stored in the cases of a method. Every method is created with at least one case and more may be added later. In fact, you will add another case to a method later on in the tutorial. Double-click the Main Method method item to bring up the only current case window for the method, case window number 1 which will be entitled "Main Method 1:1". The notation "1:1" means that this is the first case of a total of one case.

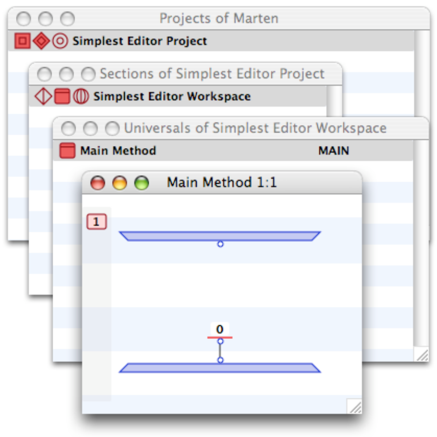
A case always has two operations, an input bar and an output bar and these are automatically supplied when you created the method. For a "main" entry point method, the input bar must have a root (the bubble below the top operation) and the output bar must have a terminal (the bubble above the bottom operation). The terminal of the output bar must supply an integer (usually 0) to the application context or operating system. To create an operation that provides this integer, double-click the terminal of the output bar.


Projects of Marten
Simplest Editor Project
Sections of Simplest Editor Project
Simplest Editor Workspace
Universals of Simplest Editor Workspace
Main Method
MAIN
Main Method 1:1
1

Double-clicking a terminal or root is a short cut to creating a constant or match operation already connected to the terminal or root. The default value of the constant or match operation is NULL.


Projects of Marten
Simplest Editor Project
Sections of Simplest Editor Project
Simplest Editor Workspace
Universals of Simplest Editor Workspace
Main Method
MAIN
Main Method 1:1
1
NULL

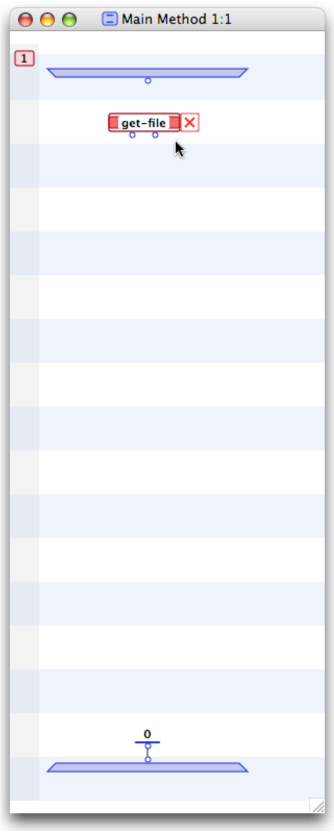
The text of the operation will be selected, so just type a 0 and you're done creating the necessary output for the main method.


Projects of Marten
Simplest Editor Project
Sections of Simplest Editor Project
Simplest Editor Workspace
Universals of Simplest Editor Workspace
Main Method
MAIN
Main Method 1:1
1
0

File manipulation coding

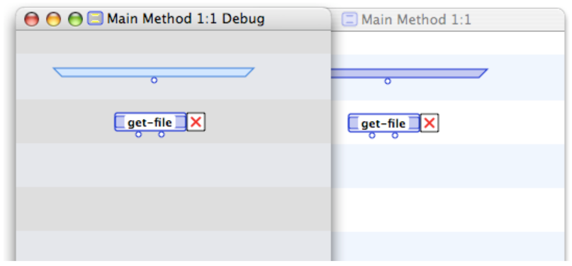
Other than the constant operation which supplies the 0, the operations needed for the text editor are all primitives. To fit them all in, stretch the case window to be a bit longer.

The first primitive we need is one that will allow us to select a file to open. The primitive that provides this functionality is the "get-file" primitive. Create it by command-clicking in the case window and renaming the operation "get-file". The operation will change style into a primitive with two output roots. In addition, the primitive operation will have a "Next Case on Failure" control annotation.

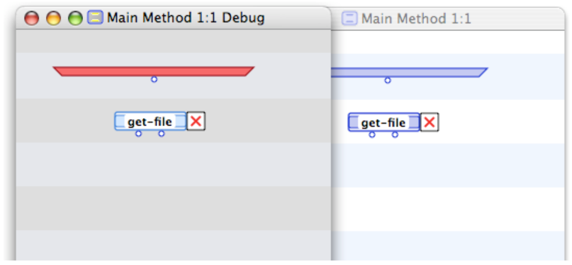


Running the method in the debugger

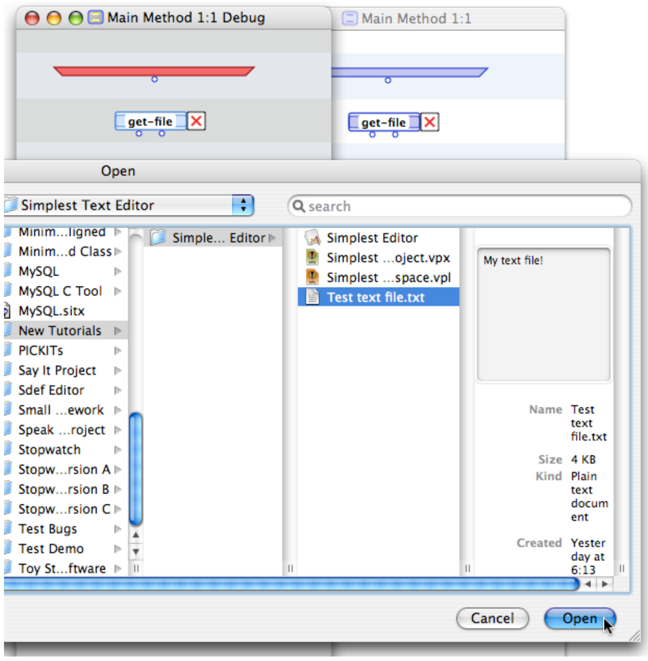
Now the best way to see what an operation does is to execute it or run it. Before you create any more code, see what the primitive does. Select "Run Method" from the Run menu or type Command-Shift-D. A new window opens, a debug window. Move the editing case window off to the side and bring the debug window for the case to the front. The input bar will be a lighter shade in the debug window than the get-file primitive operation. That indicates that the input bar is the next operation to be executed or "pending".



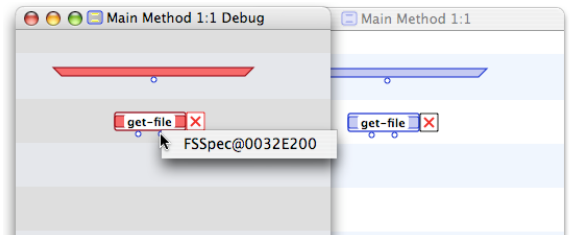
Execute the input bar by hitting the Return key or selecting Step from the Run menu. The input bar is now colored red indicating that that operation has been executed. The get-file primitive is now ready for execution.



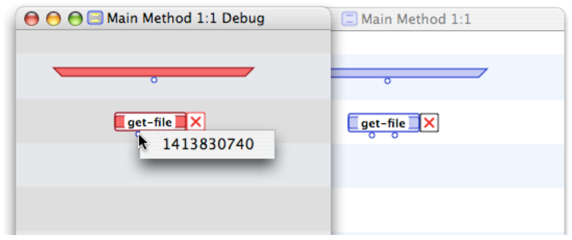
Execute the get-file primitive by hitting the Return key. Before it turns red, an Open dialog appears. This is what the get-file primitive does, it allows you to select a file. Select the "Test text file.txt" file and click the Open button.



The get-file primitive now turns red indicating that it has been executed and the roots contain data. To see just what data is contained in the right root, mouse-down on the right root. A small contextual menu appears with the value of the root displayed. In this case the value of the root is an external block object containing a MacOS X structure called an "FSSpec" which is short for "File System Specification" structure.

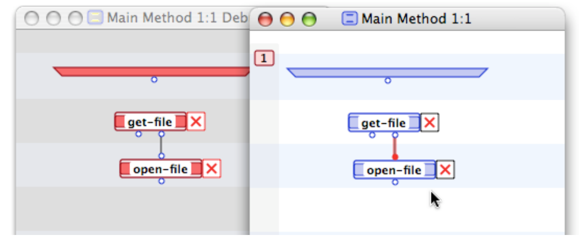


Now mouse-down on the left root. The contextual menu appears with an integer. This integer represents a four character code representing the file type of the selected file. In this case, the integer represents 'TEXT' which is the four character MacOS X file type for a plain text file.

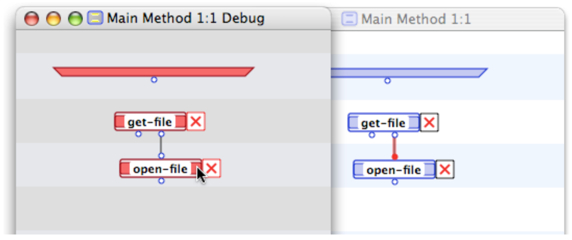


Opening the file

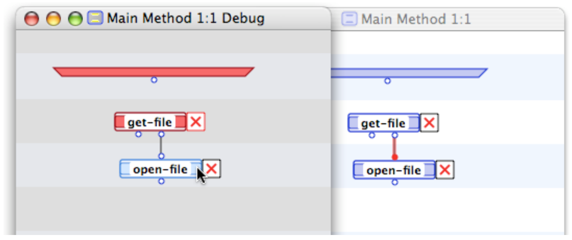
The basics steps to dealing with a file are to get a file specification for the file by either selecting one already on the disk or creating one. Once you have the file specification, you typically open the file, read the contents, and then close the file. This is exactly what we are going to do in this phase of the project. Open the file by bringing the editor window to the front and creating an open-file primitive operation. The newly created operation appears in the debug window as well and is colored red indicating that it has been executed.



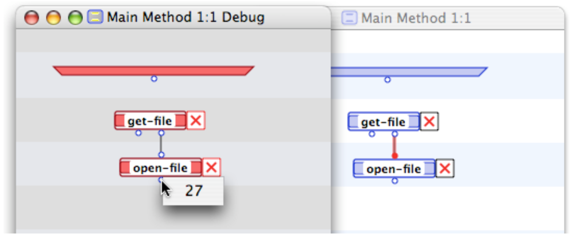
This red color is unfortunately misleading. The operation is technically "executed" because it has valid (but NULL) outputs. However this is not really what we want, so we will go back and force the primitive to execute with the file specification as input. Bring the debug window to the front



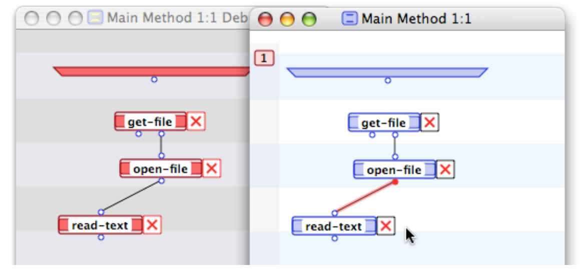
and then command-click on the open-file primitive. This forces the debugger to "roll-back" execution so that the "clicked on" operation is now the next one to be executed. The color of the primitive now changes to "pending".



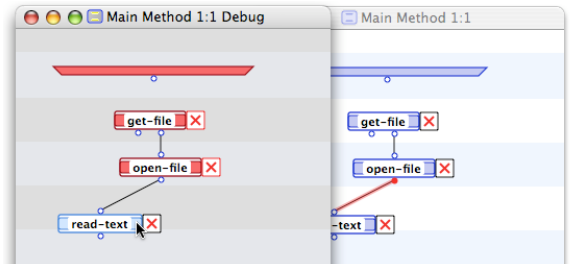
Now step forward (either select Step or hit the Return key) and examine the value of the output root. It is an integer called the "file number". It is this integer we need to supply to the subsequent operations.



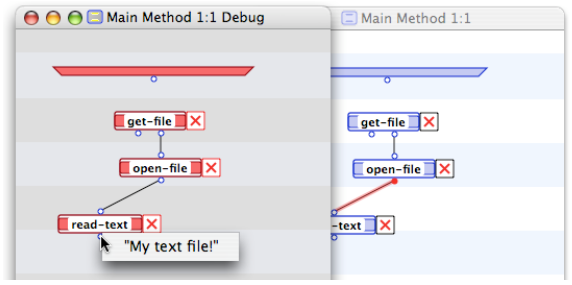
The next step is to read the contents of the file. Bring the case window to the front and create a read-text primitive. Connect the root of the open-file primitive to the terminal of the newly created read-text primitive.



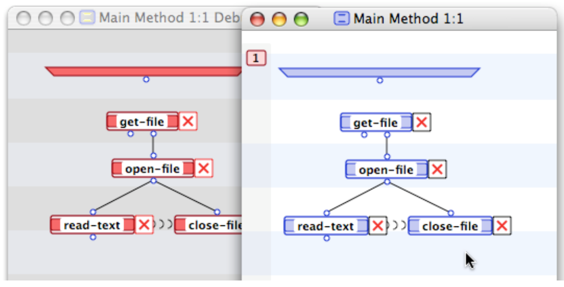
Now repeat the business of command-clicking on the read-text primitive to schedule it for execution



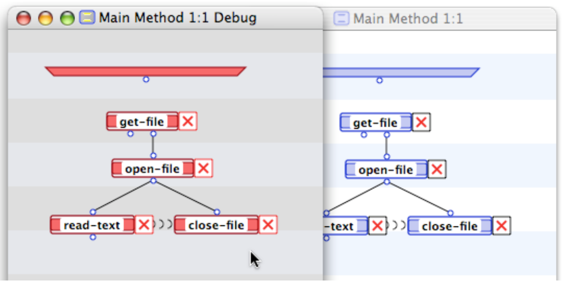
and then execute it. Examine the value of the read-text root. It is the text of the selected file.



The final step is to close the file. Add a close-file primitive to the case window and connect its terminal to the root of the open-file primitive. Now we always want to read the contents of the file before we close it. To enforce this temporal order, create a synchro from the read-text operation to the close-file operation by selecting the read-text operation, then holding down the Option key while clicking on the close-file operation.



Now return to the debugger and do the usual business of command-clicking the close-file operation to make it pending and then hitting the Return key to execute it.

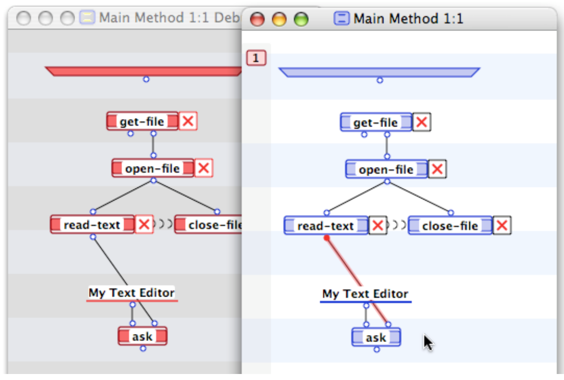


You're done with the first phase of the editor. You have selected a file to open, opened it, read its contents, and then closed the file. The contents are now available for you to do what you want with them.

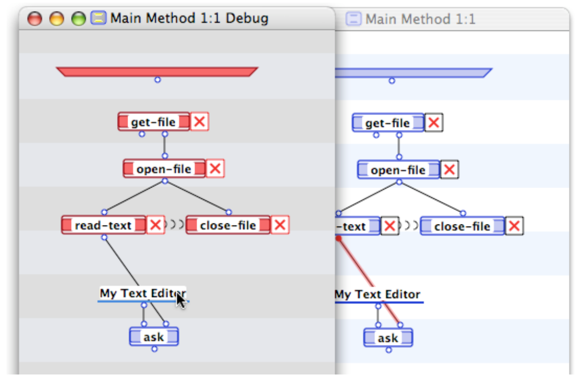
The text editor

Now that you have the contents of the text file, it is a simple matter to open a minimal text editor. Marten ships with a primitive that opens a window that can server as a basic text editor. This is the "ask" primitive. Add an ask primitive to the case window. When you do, the primitive operation will not have any terminal bubbles at the top of the operation. The ask primitive doesn't require any inputs, but it can accept at least two optional inputs. If you create one input, any text supplied to that input will be used as the "comment" text at the top of the window. If you create a second input, any text supplied to that input will be used to initially populate the contents of the text editor window which you can then modify to your heart's content.

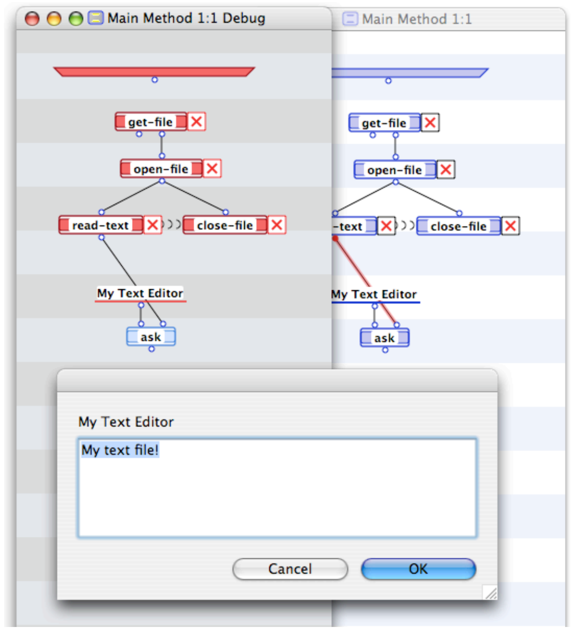
We want these two additional inputs, so command-click just above the ask primitive operation. An terminal bubble is created. Move it to the left and command-click again to create the second input. Double-click the left terminal to create a constant operation connected to it. Modify the text of the constant operation to "My Text Editor". Next connect the output of the read-text primitive to the right terminal of the ask operation. When you are finished, the case window should look like:



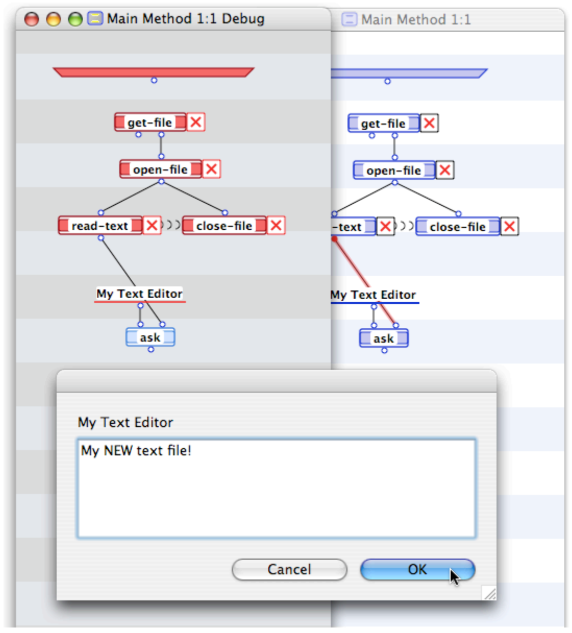
Now we want to execute the ask primitive but if we do that, the left terminal will have a NULL input. We need to back up and execute the constant operation first so that the comment text will be created. Bring the debug window to the front and command-click on the constant operation.



Hit the Return key to execute the constant operation. Now its root will supply the appropriate text to the ask primitive. Hit the Return key again to execute the ask primitive. A simple dialog window opens with the comment text in the upper left and the contents of the text file displayed in the editor.



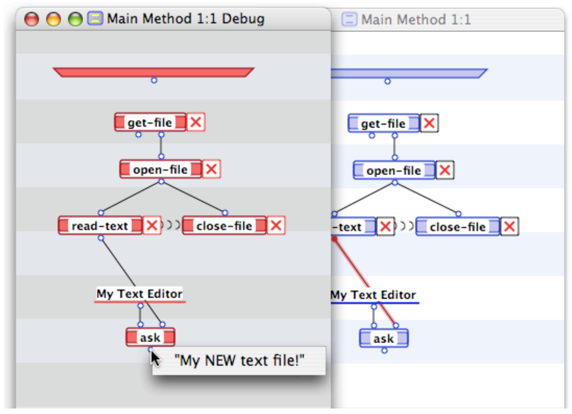
Modify the initially displayed text by inserting the work "NEW" in the phrase. Save your modifications by clicking on the "OK" button.



My Text Editor

My NEW text file!

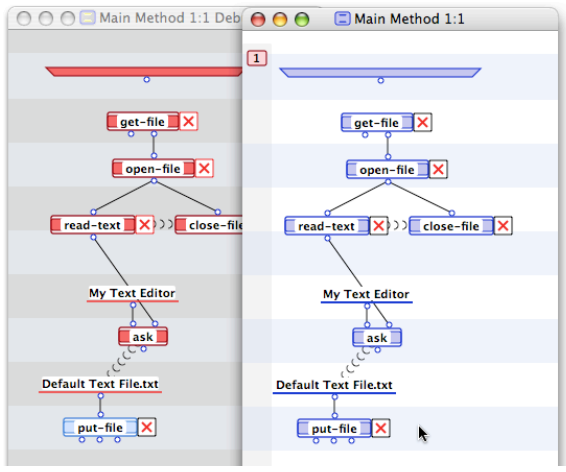
The ask primitive is now colored red in the debug window indicating that it has been executed. Mouse down on the root of the ask primitive to see that the new text has your modifications.



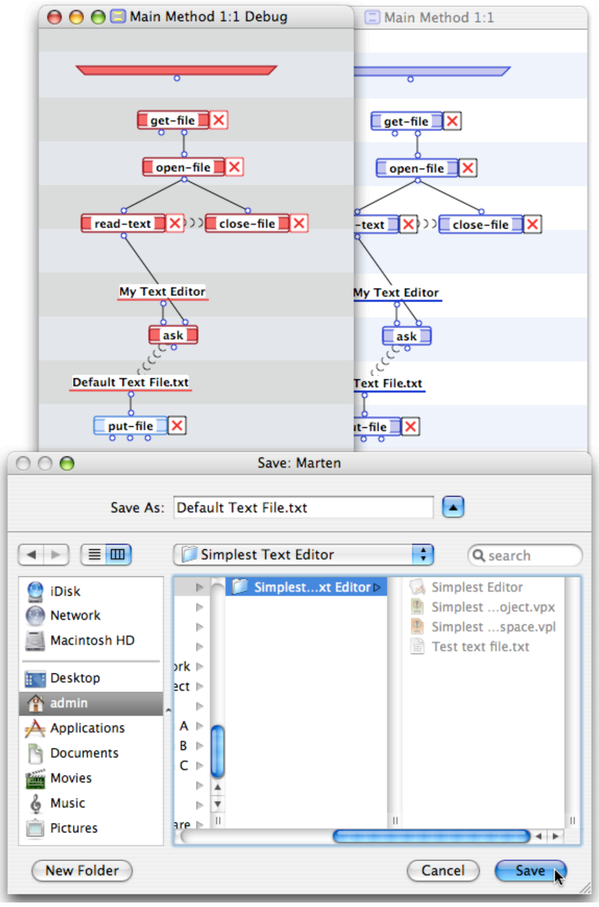
"My NEW text file!"

Saving the modified text

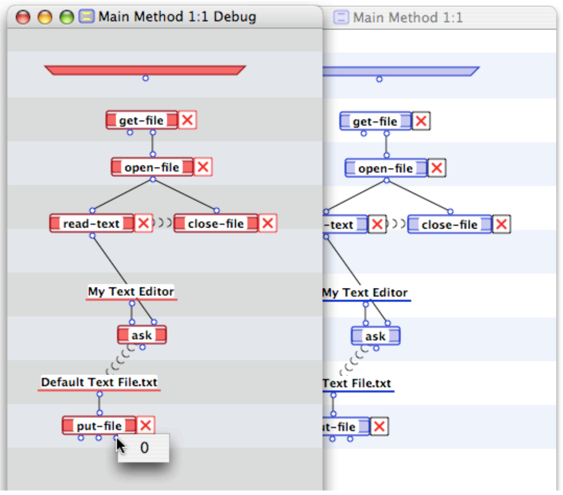
There is no point modifying text if you can't save it, so now we add that functionality. The first step is to provide a dialog that allows the user to decide where to save the file. Add the following two operations; a constant operation with the value "Default Text File.txt" and connect it to a new "put-file" primitive operation. Then command-click the constant and execute it so that the put-file operation is pending.



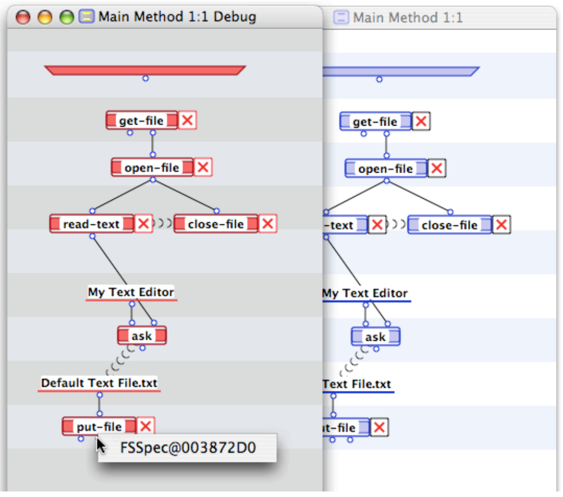
Now hit the Return key to see the put-file primitive in action. A Save dialog opens up allowing you to specify the name of the file and its location. The text supplied by the constant operation is used as the default name for the file. Just accept the default name and save the file in the project directory.



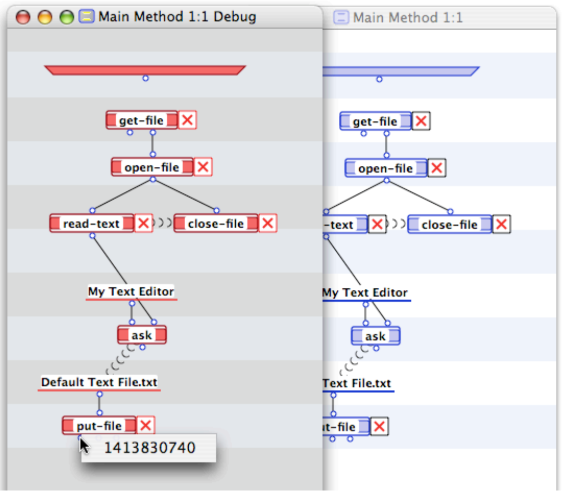
The put-file primitive has now been executed and we can see what data has been generated. Clicking on the right root of the primitive shows that an integer with value 0 was created. This is the "script code" and is needed as an input for the "create-text-file" primitive we will create later on.



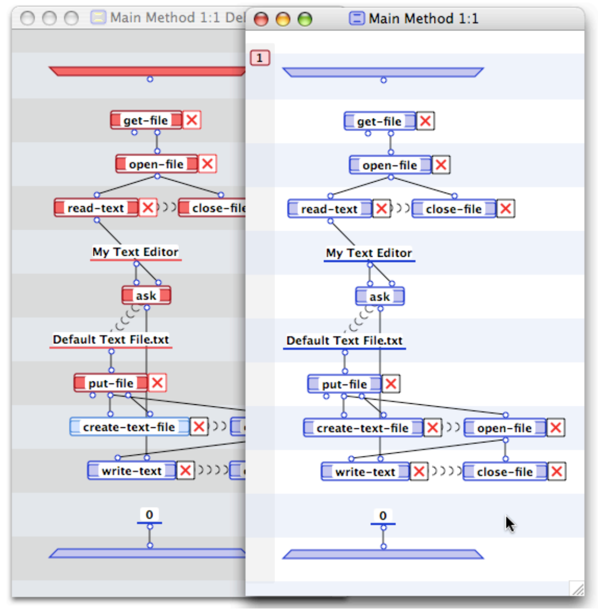
The middle root of the primitive shows that a file system specification structure was created. This structure is needed to create a file and then just as before, to open it.



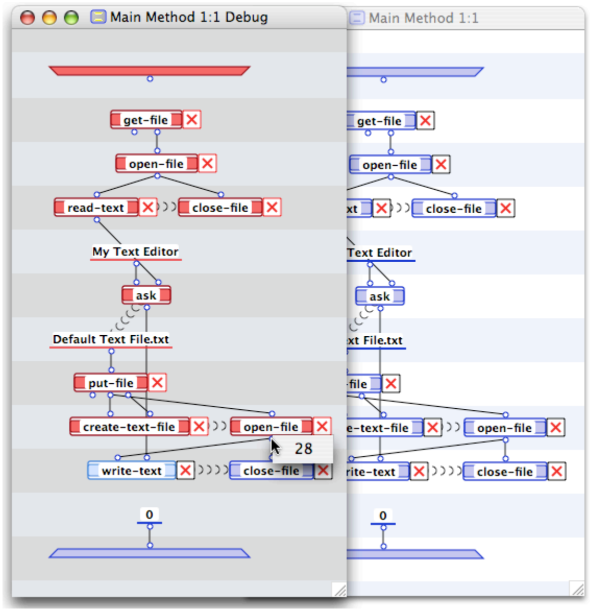
The left root of the primitive contains the value of the file type. Again, this is a text file.



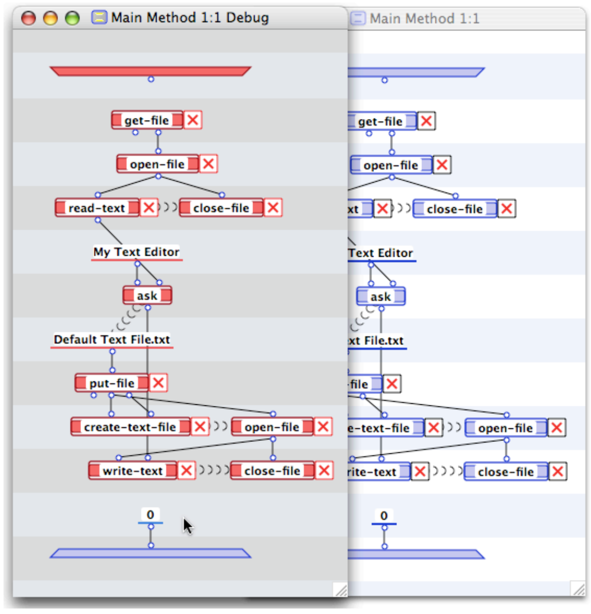
The file manipulations we need to perform are just the same as before except that we need to actually create the file on disk. This is done by using the create-text-file primitive. Go ahead and add the four primitives displayed below and then command-click on the create-text-file primitive to put it in the pending state.



With the debug window in the foreground, hit the Return key to execute the create-text-file primitive and then hit the key again to execute the open-file primitive. We see that the operation was successful and that the new file number is 28.



Continue to execute the remaining file manipulation operations.

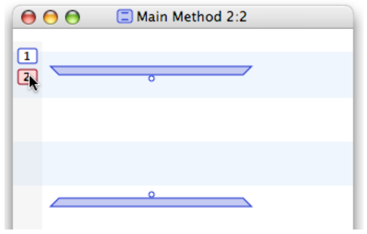


and then continue to hit the Return key to execute the remaining constant and output bar operations. The debug window will close as the method has been successfully run.

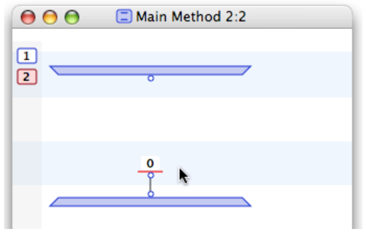
Creating the next case

All the primitive operations created have been annotated with a "Next Case on Failure" control (the square box with the X inside). This means if the operation fails for any reason (for example, clicking "Cancel" on the modal dialogs), execution of the method will skip to the next case. However there is no next case! You must make one or your final standalone text editor application will crash when that happens.

To create a new case, command-click on the "case dock". A new case icon will appear and the code of the new case will be displayed in the editing window.



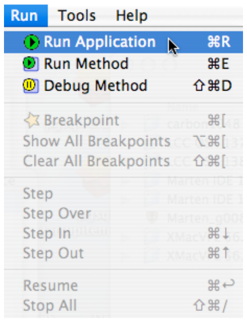
Create the standard output for a main method just as you did earlier.



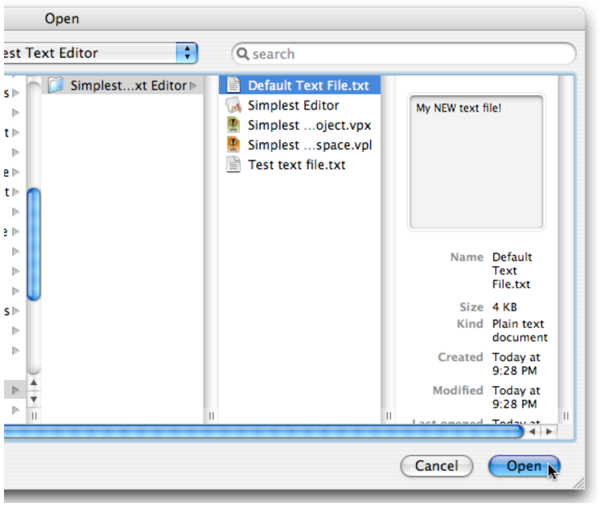
Your coding is finished and your application is ready to run.

Running the application in the development environment

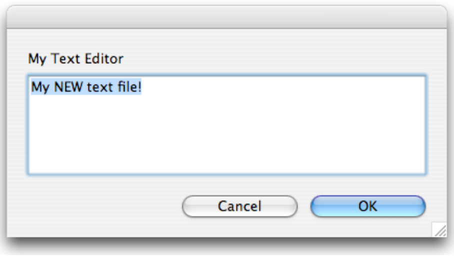
You should run your application to see if it works as you expect. To do this, select the Run->Run Application command or hit Command-R.



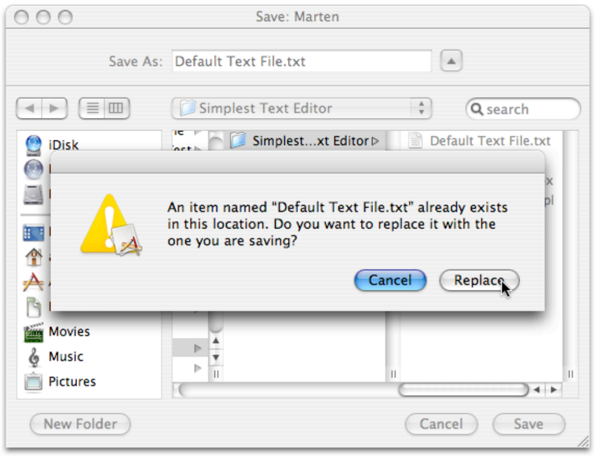
An Open dialog will appear. Select your newly modified test text file and click Open.



The dialog will close and the contents of the text file will appear in a new "My Text Editor" window.



Modify the text however you want and then click the OK button. A Save dialog window will open. Accept the suggested name and click the Save button. A "Replace?" dialog will then open. Click the Replace button.



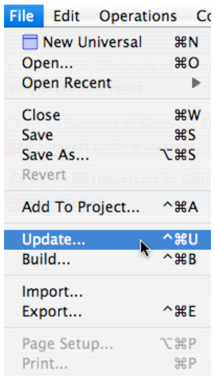
An item named "Default Text File.txt" already exists in this location. Do you want to replace it with the one you are saving?

The application will quit. Open the file with TextEdit and check to see if your modifications have been saved. Everything worked as expected so you are now ready to create a standalone version of the Simplest Text Editor application.

Creating a standalone application by updating the project application

Marten provides two very different ways to create a standalone application. Each way creates a specific type of standalone application. One way turns your project application (the "Simplest Editor.app" you saved in the second step of "Creating the project") into a standalone version of the application that was run by the Marten IDE. This type of application is an interpreted one. In other words, the application consists of an interpreter and a "giant" Marten project file that contains all of your sections. When the application is launched, the interpreter is started, loads the Marten code, and then runs it.

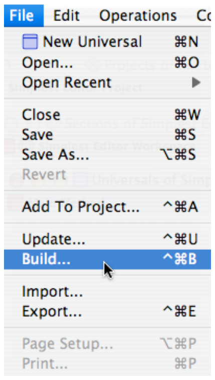
Since this process simply converts your pre-existing project application into a standalone application, you merely "update it". Select the "Update..." command under the File menu.



Viola! Your Simplest Editor project application file in your project folder is now a standalone application. Don't believe me? Just double-click it.

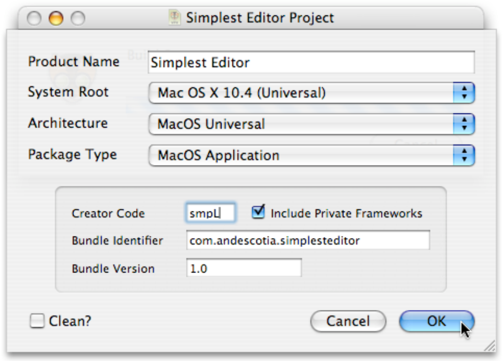
Creating a native compiled universal application

The other kind of standalone application that Marten can create is a more typical "native" one. This means that the Marten code is converted into machine language that the operating system can load and run. This type of application will generally run much faster than the interpreted version but if you have any bugs in your code, this version will crash rather than shutdown gracefully. This conversion into machine language is called "compiling" (and "linking"). Create this type of application by selecting the "Build..." command under the File menu.



A dialog window will open giving you some choices regarding your application. You should make the changes illustrated below. The name of the application will now be "Simplest Editor", it will require MacOS X 10.4, it will be a universal application that will run on PPC and x386 machines, it will be a standard GUI application (as opposed to a "command line" one), the creator code is 'smpL' ("simple"), it will include all the libraries needed to run inside its application package, the application bundle will be identified by the phrase "com.andescotia.simplesteditor", and finally the version of this application bundle is just "1.0". In addition, since this is the first time you will have generated any C source code and machine language files, you do not need to "clean" them away.

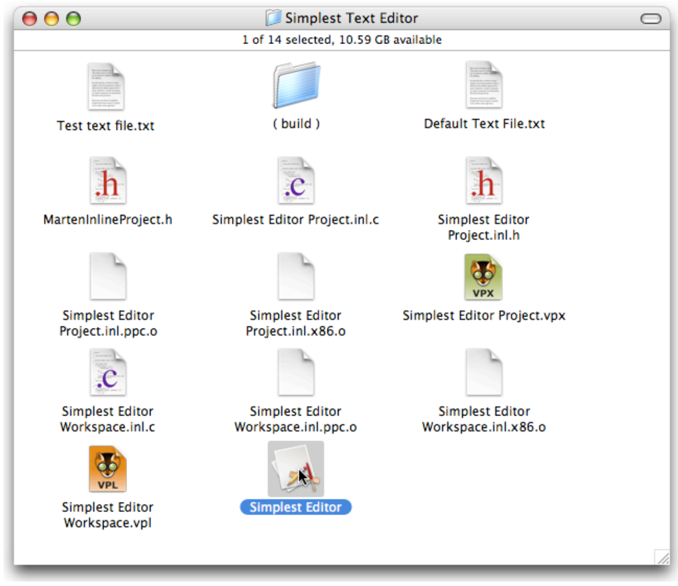
Click the OK button to start the process.


Simplest Editor Project

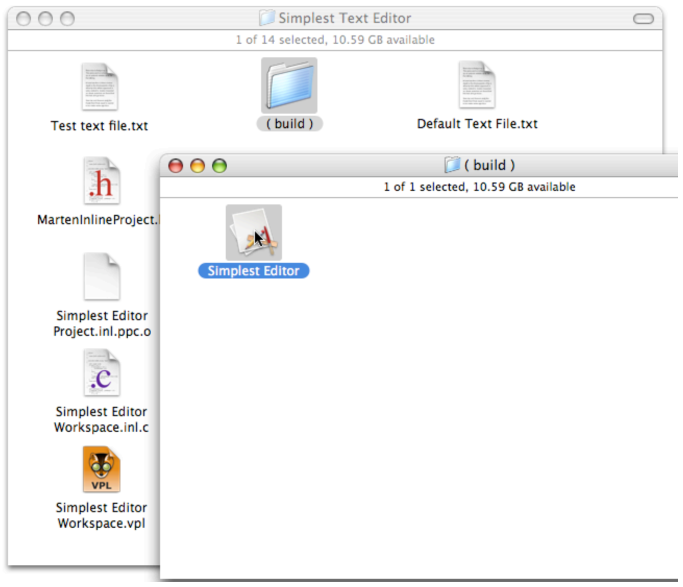
Product Name: Simplest Editor
System Root: Mac OS X 10.4 (Universal)
Architecture: MacOS Universal
Package Type: MacOS Application
Creator Code: smpL
☑ Include Private Frameworks
Bundle Identifier: com.andescotia.simplesteditor
Bundle Version: 1.0
☐ Clean?

Launching the various applications

Now that you have made two different kinds of your application, where are they? The updated version is of course your project application which you saved into your project folder.


Simplest Text Editor
1 of 14 selected, 10.59 GB available
Test text file.txt
(build)
Default Text File.txt
MartenInlineProject.h
Simplest Editor Project.inl.c
Simplest Editor Project.inl.h
Simplest Editor Project.inl.ppc.o
Simplest Editor Project.inl.x86.o
Simplest Editor Project.vpx
Simplest Editor Workspace.inl.c
Simplest Editor Workspace.inl.ppc.o
Simplest Editor Workspace.inl.x86.o
Simplest Editor Workspace.vpl
Simplest Editor

Notice all the new files in your project folder, generated as a result of the native application build process. There are the C source code files like "Simplest Editor Project.inl.c" and the machine language files like "Simplest Editor Project.inl.ppc.o". The native application that was generated is placed into a new subfolder named "(build)".


Simplest Text Editor
1 of 14 selected, 10.59 GB available
Test text file.txt
(build)
Default Text File.txt
MartenInlineProject.
Simplest Editor Project.inl.ppc.o
Simplest Editor Workspace.inl.c
Simplest Editor Workspace.vpl
(build)
1 of 1 selected, 10.59 GB available
Simplest Editor

Conclusion

If you made it this far, then you have successfully created a very minimal plain text editor. You have seen how to use the most important of the Marten file manipulation primitives and how to use one of the most flexible of the Marten GUI primitives. But you can go further. One suggestion is to use the "Operations to Local" and "Local to Method..."commands (found under the Marten IDE Operations menu) to group some operations into a "Get Text File Contents" universal method and some other operations into a "Save To Text File" universal method to simplify your code.

If you do that, you could then create a text editor that doesn't just quit right away when you are done with your modifications. To implement this functionality, you could create a local operation in your main method that nominally repeats forever. Inside that local you could then use the "answer" or "select" primitives to ask whether the user would like to edit a file or quit the application. If the answer is "edit", then you would proceed to "Get Text File Contents" and so on. But if the answer is "quit", then you would terminate the repeating local operation to return to the main method and exit.