

Mojo™

Programmers Quick Start



Copyright© 1996. Penumbra Software, Inc. All Rights Reserved.

Chapter 1

Mojo Programming in the Coder

Overview

Mojo has two primary aspects, a designer and a coder. The designer is a high-level drag-and-drop GUI builder while the coder is the back-end development environment for adding complete functionality to your applications. While it is possible to add functionality to a Mojo-created Java program without touching a line of code using Actions (see the Getting Started document), the coder makes Mojo truly robust and gives programmers the ability never to be limited by their environment.

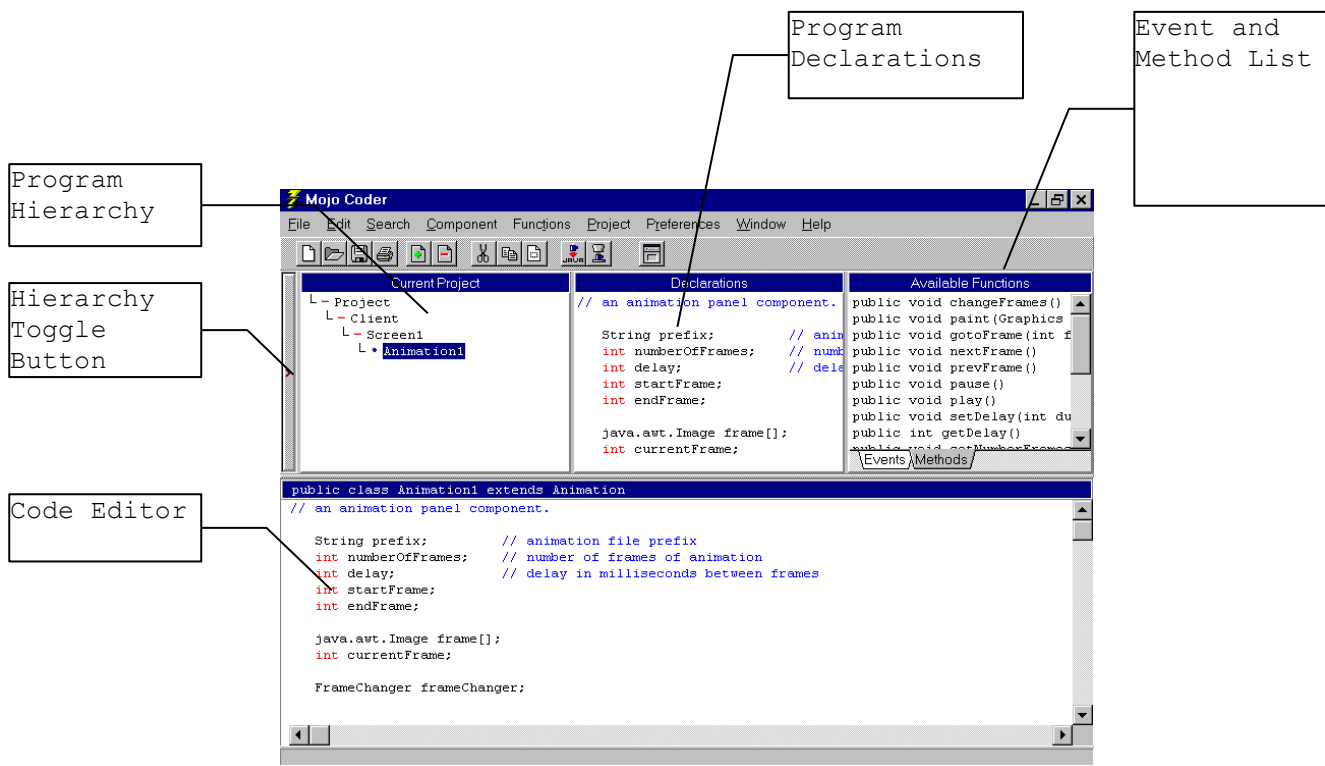
This chapter provides a guide into the coding aspect of Mojo. It goes beyond the basic actions of drag-and-drop and provides an overview of how to make useful applets and applications. This guide is intended for programmers who have become familiar with the basic Mojo environment and want to move into Java programming.

Coder Diagram

The following is a basic diagram of the Coder as it appears during startup. It is primarily a three panel class browser with a coder editor.

You can use the right mouse button in the various panels to bring up special menus for each panel. For instance, a right-click in the Event and Method List panel will bring up the override list for the selected class.

This diagram will be referenced by other sections of this document.



Dual Hierarchies

Mojo uses two distinct hierarchical classifications -- a class hierarchy which is arranged into packages and an container hierarchy which organizes your project.

The Class Hierarchy (maroon colored), also referred to as the Component Hierarchy, is where all Java classes and available components reside. It can be accessed by pressing the Hierarchy Toggle button found on the left-hand side of the browser windows. The Class hierarchy is arranged by packages. For instance, the Button class is found in:

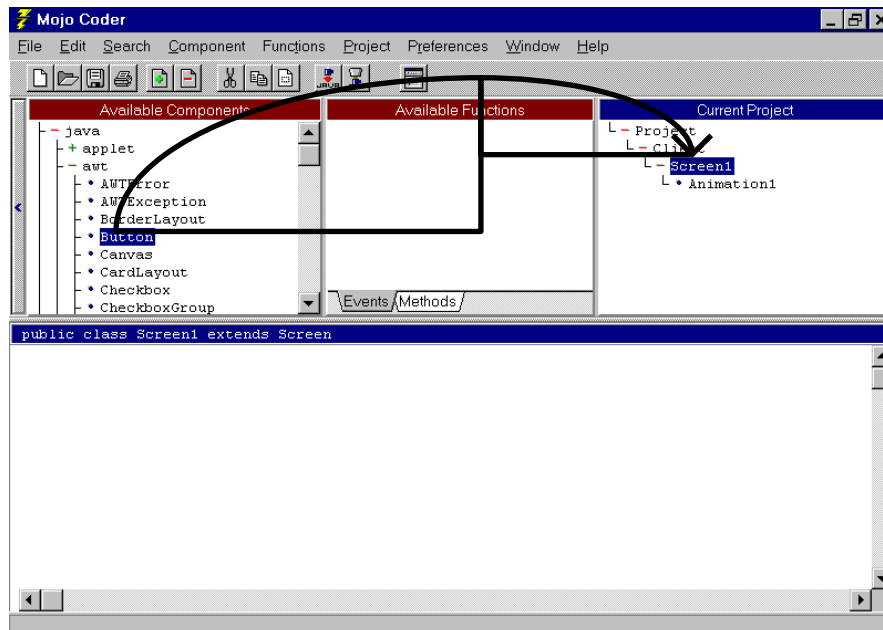
```
java
```

awt
Button

The Project hierarchy (navy colored) is where all the instances of class are stored. Programming in Mojo is done by changing attributes of instances. Associated classes are automatically generated for each instance. For example, you can change the behavior of a particular button, i.e. Button1, by overriding its action event or its paint method. Unlike raw Java, you do not need to create a new class for every component for which you wish to have a particular behavior. The programmer can concentrate purely on extending the functionality of what they have inserted into their project.

Adding a Component in the Coder

Slide open the Class hierarchy (maroon) by pressing the Hierarchy Toggle button. Browse and find the component in the Class Hierarchy which you wish to make an instance of. For example, if you wanted to add an instance of a Stack, you would browse through java -> util -> Stack and click on Stack. Now, click on the component in the Project hierarchy where you want to add the instances as a child of. Drag-and-drop the Stack component into the Project. A dialog will appear asking for the instance name. After you provide an instance name and click on **Ok**, an instance of Stack with the name you provided will be added to your project.



Removing a Component

To remove a component from either hierarchy, click on the Component in the name in the Available Components or Current Project panel. Then select Remove Component in the Component Menu. A short-cut is to use the right mouse button in the component or project panel for the pop-up menu. Removing a component will remove a component and all of its subcomponents as well as all subclasses of the component.

Overriding a Function

Synopsis: Right-click on the Method and Event list to bring up an override list.

To override a function in Mojo, select the component by clicking on it in the hierarchy. Then, press the right mouse button in the event and method list panel. A pop-up menu will appear. Select the Override option to bring up a function override list. If the override list is short, another pop-up menu will appear with all function headers which you can override (grouped by inherited components). If the override list is long, a separate dialog box will appear showing all inherited functions. Double-clicking on a header will override the function.

When a function is overridden, a skeleton of the function is loaded in the code editor.

Adding a Function

Synopsis: Add a function to a component using the Add Event or Add Method options under the Functions menu.

To add a function, select the component which you want to add the function to. Then, click on the Add Event or Add Method option under the Functions. Whether you are adding an event or method depends on the which option is selected on the tab under the Method and Event List. When you add a method or event, you have the option to make the method public, private, or protected using radio buttons in the Add Function dialog box. A shortcut to add a function is to right-click in the Available Functions pane.

Removing a Function

Synopsis: Remove a function from a component using the Remove Event or Remove Method option under the Functions menu.

To remove a function, select the function header in the Method and Event List. Select the Remove Event or Remove Method from the Function menu to remove the function. A shortcut to add a function is to right-click in the Available Functions pane.

Updating the Code

Synopsis: *Use <CTRL>-U to update your code.*

Mojo uses a Smalltalk-like browser which has two important code editing commands - update and revert. When you type code into Mojo, it is temporarily stored in a buffer. In order for Mojo to incorporate what you have typed, you need to update your code by pressing <CTRL>-U in the Code Editor. Clicking away to another panel or issuing a command also updates the code. Until you update the code, you have the option to revert your code back to the last update by pressing <CTRL>-R in the Code Editor.

Changing Component Declarations

Synopsis: *Use the component's declarations to add instance variables and class variables.*

When you select a component by clicking on the component name in the Program or Component hierarchy, the components declarations appear. A component's declarations store instance and class variables. You can directly type in the declarations in the Code Editor. After you update the code, the declarations will appear in the Program Declarations pane. *Note: Do not put the { or the } in the declarations, they are automatically generated by the Mojo.*

Component Referencing Model

Synopsis: *Use the Mojo reserved words **parent** and **applet** to reference other components.*

Referencing components in Mojo is significantly easier than using pure Java. There are two primary keywords used for accessing parts of your applet:

```
applet    <- currently running applet
parent    <- parent of the object
```

For instance to find out where the applet was started from, you could call:

```
applet.getCodeBase();
```

For the given hierarchy below, you can reference another components methods and instances variables using the parent-reference model:

```
--+
|
+- Panel1
|
|   +- Button1
|   |
|   +- Button2
```

In the action event of Button2, you can call the action event of the Button1 with:

```
Button2:
    public boolean action(Event evt, Object what)
    {
        parent.Button1.action(evt, what);

        return(true);
    }
```

Using parent, you are able to access all parts of the Project hierarchy.

Initializing a Component

*Synopsis: Place initializations in the **void initialize()** method.*

In an visual object environment like Mojo, certain operations need to be done in the constructor. This is accomplished by the Mojo code generator. The code generator creates a special constructor which sets the default property values and handles the Component Referencing Model. However, there are instances when a programmer might wish to have code which is run as part of the initialization of a component. This is accomplished using the **void initialize()** method. Any code added to the initialize() method of a component will be run after all other initializations of the component. You can add the **void initialize()** method to a component by override it in the override list of a method or by using the Add function option under the Functions menu in the coder.

Importing a Package

Synopsis: Use the Attributes option under the Component menu to add or remove import statements for a particular component.

The common Java model for importing a package is done on a global level. In Mojo, packages can be imported on a component by component basis. Select the component to add an interface for. Select on the Attributes option under the Components menu. This will bring up the attributes dialog box. Click on the Imports tab. To add a package, click on the Add button. To remove a package, select the package to remove from the list and click on the Remove button. Click on the Attribute dialog's Ok button to accept all changes for the component.

Adding an Interface to a Component

Synopsis: Use the Attributes option under the Component menu to add or remove interface statements for a particular component.

Component can be made to implement a particular interface by adding an interface using the Attributes dialog (see Importing a Package). Interfaces are found under the Implements tab in the Attributes dialog. In the declaration **public MyApplet extends Applet implements Runnable**, the interface to add is **Runnable**. Java allows you to add more than one interface for each component.

Regular Java Declarations

*Synopsis: Use the **Project** declarations to enter all straight Java declarations.*

Most Java programs are arranged using "flat files." Mojo makes programming simpler by arranging the program based on hierarchies. This divides a program into small functional units which are easier to manipulate. To allow programmers absolute flexibility, Mojo allows them to type in straight "flat-file" Java code.

Clicking the Project node in the Program hierarchy brings up the declarations for the entire project. The Code Editor now acts as a flat file editor for Java code. Class and Interface declarations as well as extra import lines can be typed into the Project declarations.

The Project declarations appear before any other Java code is generated.

Compiling the Applet

Synopsis: Compile the Applet using Compile Applet option under the Project menu.

Compiling an applet creates a .java file from the current Mojo project and compiles it using the compiler specified in the Project Preferences. Compile the Applet using the Compile Applet option under the Project menu.

Compiler Error Messages

Synopsis: If an error in the code is found during compilation and error message will be displayed in the Coder Status bar.

Compile errors will attempt to jump to code where the error occurred. Since Mojo is part a visual environment, some error messages do not correspond to functions, method, or declarations. In these cases, an error dialog will appear showing all error messages. Double-clicking on an error jumps you to the corresponding error location in the .java source file.

Previewing the Applet

Synopsis: *Preview applets using Appletviewer.*

The compiled applet can be viewed in using the Preview Applet option under the Project menu. This will bring up Appletviewer setting the appropriate CLASSPATH. You can change the location of the appletviewer in the Project Preferences dialog (use the Project option under the Preferences menu).

Previewing the Applet using Netscape's Navigator

Synopsis: *Preview applets using Netscape's Navigator or any other browser by changing the Netscape Browser path in the Preferences dialog.*

Use the Project option under the Preferences menu to bring up the Preferences dialog. Under the Browser tab, you can browse for your Netscape Navigator. After you have selected a browser, select Preview in Browser under the Project menu. This will check the Preview in Browser menu option. Now, when you go to Preview Applet, it will preview using the Netscape Navigator. This also applies for Running an Applet as well. To revert to using the Appletviewer, uncheck the Preview in Browser menu option by selecting it again.

Making a Component for Reuse

Synopsis: *Make components out of parts of your project for the ultimate in reuse.*

As you build a Mojo project, each part of the project can be made into a component for future reuse. If the component made is visual, it will appear in the User-Defined tab of your component palette of the Designer as a drag-and-dropable component. Otherwise, the component will be added to the user package in the Class hierarchy.

To make a component, either select it in the Designer or in the Program Hierarchy of the Coder. Then select the Make Component option from the Component menu. A dialog box will appear asking for the name and default package for the new component. *Note: the name of the new component must have a different name than the original component.*

If you make a component from a panel which contains other subcomponents, all the subcomponents will be made into components for future use as well; however, only the master component will appear on the palette. For example, an interface for a VCR control which has a reverse and forward button will create one VCR component which will include both the reverse and forward buttons which are accessible in the Class hierarchy.

Output Directory

Synopsis: All *.class*, *.java*, *.HTM*, and *resources* are stored in the output directory of the current project.

Mojo makes creating applets and applications simpler by gathering all resource you use while developing into an output directory. The output directory is a special directory which branches off of where you saved the file and named <projectname>.output.

For example, a project named **MyApplet** stored in the directory **c:\applets\special**, creates the following directory structure:

```
c:\applets\special\MyApplet.prj      <- project file
c:\applets\special\MyApplet.def      <- project definition file
c:\applets\special\MyApplet.output\  <- project output directory
```

To put an applet onto a website, transfer all the files in the output directory, except *.BMP* files and the *.java* source, to the site. Depending on whether or not third party components are used, you may need to transfer additional *.class* or *.zip* files to your site as well. Please refer to special documentation on third party components to see which *.class* or *.zip* files need to be transferred to make a particular component work on the net.

.java Source File

Synopsis: *A flat .java file is produced during compilation of your applet or application and is found in the output directory.*

Mojo generates a complete .java file when it compiles or runs your applet or application. This source file can be found in the output directory.

Common uses for the .java is to use it in a third party editor, or for complex debugging or understanding how Mojo generates your Java program.

Mojo Directory Structure

Synopsis: *A listing of the Mojo directory structure.*

The following directories are assumed to branch off of the Mojo install directory:

<i>mojo_install_path</i>	Mojo executable
\images	.gif image library
\java	JDK
\lib	Mojo library files
\plugins	3 rd party component definition files
\classes	3 rd party component classes
\user	User-defined component (Make Component)
\samples	Sample projects
\sounds	.au sound library
\docs	Off-line documentation
\myproject	Temporary project until save

Mojo Package Structure

Synopsis: *Mojo components are arranged into a package view in the Class hierarchy.*

Mojo packages are arranged into abstract packages. Java classes are arranged into a package called *java* and organized in the same way as the package import specification of the JDK. Mojo components are in the *mojo* package. Third party components are placed in the *plugins* package. User-defined components (created with Make Component) are placed in the *user* package.

A package/component/subcomponent model is used to organize components. For example, a component made by a user called *MyVCRComponent* which is under the *mediaplayers* category will be stored in the following manner:

```
user
  MyVCRComponent
    subcomponents
      myvcrcomponent
```

ForwardButton
ReverseButton
PauseButton
PlayButton
StopButton

Note: all subcomponents for the VCR control are stored under subcomponents in another package heading named by the same name.

Special Note: Package names should be all lowercase by convention, and Class names should start with a capital letter.

Using Third Party Compilers

Synopsis: Mojo allows third party compilers to be plugged in using the Project option under the Preferences menu.

Mojo allows the programmer to select any Java compiler of their choice. Currently, two compiler error models are support - Sun's javac.exe and Symmantics' sj.exe. The compiler defaults to Sun's javac. To change the compiler, select the Project option under the Preferences menu. The default Java compiler and CLASSPATH can be changed in the Compiler tab.

Chapter 2

MOJO Third Party Components

Overview

The following is a listing of third-party components which are included in the initial release of Mojo. A brief description of each component and their properties is provided. For further documentation, refer to the *docs* directory of your Mojo installation.

MarqueeLights

The MarqueeLights component puts a user-defined .gif file inside a moving marquee of lights. Use this to visually enhance or bring out any pictures you may want to appear in your applet/application.

Properties:

- **GIF Filename** :: reflects the file you would like within the MarqueeLights component.
- **Top** :: Y location of the top side of the component.
- **Left** :: X location of the left side of the component.
- **Height** :: the height of the component in pixels.
- **Width** :: the width of the component in pixels.

TickerTape:

The TickerTape component allows the developer to place scrolling text in arbitrary places within the applet/application.

Properties

- **Jump Size** :: the number of pixels you would like the text to move at a time.
- **Rest Time** :: the amount of time (in milliseconds) between the jumps.
- **Background Color** :: the color you would like the TickerTape background to appear.
- **Foreground Color** :: the color you would like the TickerTape foreground to appear.
- **Update Time** :: the amount of time (in milliseconds) the applet/application will wait before checking a URL for newly supplied text.

*NOTE: This property only applies if a URL is defined.

**NOTE: a Zero(0) value indicates that a URL should only be checked once.

- **URL** :: If this is specified, the applet/application will check the URL to get a message from. If this is left blank, you should enter something in the message property box.
- **Message** :: Enter the text here you would like to be displayed. If text is entered here, the URL property is considered to be empty.

- **Link-To** :: Input a URL to be linked to when the TickerTape is clicked on.
- **Top** :: Y location of the top side of the component.
- **Left** :: X location of the left side of the component.
- **Height** :: the height of the component in pixels.
- **Width** :: the width of the component in pixels.
- **FontName** :: the name of the Font you would like text displayed in.
- **FontStyle** :: the style of the Font you would like text displayed in.
- **FontSize** :: the height in pixels of the Font you would like text displayed.

Blink

The Blink component allows the developer to place blinking text in arbitrary places within the applet/application.

Properties

- **Text** :: the text you would like to be displayed blinking.
- **Speed** :: the rate at which the text blinks.
- **Top** :: Y location of the top side of the component.
- **Left** :: X location of the left side of the component.
- **Height** :: the height of the component in pixels.
- **Width** :: the width of the component in pixels.

Gauge

The Gauge component is a speedometer-like progress meter with many configurable options.

Properties.

- **Minimum** :: defines the minimum value of the Gauge.
- **Maximum** :: defines the maximum value of the Gauge.
- **CurrentValue** :: the value that the Gauge will reflect at startup.
- **Warning Percentage** :: The percentage at which the "Warning Zone" will begin.
- **Critical Percentage** :: The percentage at which the "Critical Zone" will begin.
- **Units** :: a string that indicates what sort of units that the Gauge will reflect.
- **Legend** :: a string that indicates what the Gauge is describing.
- **Internal Color** :: the color that will appear inside of the Gauge.
- **Background color** :: the color that will appear outside of the Gauge.
- **Top** :: Y location of the top side of the component.
- **Left** :: X location of the left side of the component.
- **Height** :: the height of the component in pixels.
- **Width** :: the width of the component in pixels.

Calendar

The Calendar component gives the developer a calendar framework to work with.

Properties

- **Top** :: Y location of the top side of the component.
- **Left** :: X location of the left side of the component.
- **Height** :: the height of the component in pixels.
- **Width** :: the width of the component in pixels.

Clock

The Clock component lets the developer put an analog clock in a applet/application.

Properties

- **Top** :: Y location of the top side of the component.
- **Left** :: X location of the left side of the component.
- **Height** :: the height of the component in pixels.
- **Width** :: the width of the component in pixels.

LEDClock

The LEDClock component lets the developer put a digital LED clock in a applet/application.

Properties.

- **Foreground Color** :: sets the color of the LEDs.
- **Background Color** :: sets the background color of the clock.
- **Top** :: Y location of the top side of the component.
- **Left** :: X location of the left side of the component.
- **Height** :: the height of the component in pixels.
- **Width** :: the width of the component in pixels.

Microline Tabs

The Microline Tabs component allows the developer to easily add a tabbed effect to an applet/application. The position of the tabs can be set through the Property Inspector. The number of Tabs can also be set through the Property Inspector. Objects within each tab must be set programatically. There are also other features which must be set programatically. Please refer to Microline's documentation for more complete information.

Properties.

- **Tabs Placement** :: an integer value which indicates what position the tabs should be placed in. (1:top; 2: bottom; 3: left; 4: right).
- **#ofTabs** :: defines the number of tabs in the tabbed panel.
- **Top** :: Y location of the top side of the component.
- **Left** :: X location of the left side of the component.
- **Height** :: the height of the component in pixels.
- **Width** :: the width of the component in pixels.

MicroLine Grid.

The Microline Grid allows the developer to easily add a grid to an applet/application. The number of columns and rows can individually be set. There is a plethora of other features that can be added programatically. Please refer to Microline's documentation for more complete information.

Properties.

- **Columns** :: number of columns in the Grid.
- **Rows** :: number of rows in the Grid.
- **Top** :: Y location of the top side of the component.
- **Left** :: X location of the left side of the component.
- **Height** :: the height of the component in pixels.
- **Width** :: the width of the component in pixels.

Microline Progress Bar

The Microline Progress Bar allows the developer to easily add a progress/status bar to an applet/application. It features a veritable cornucopia of properties.

- **Bottom Shadow** :: the color of the shadow around the bottom of the progress bar.
- **Top Shadow** :: the color of the shadow around the top of the progress bar.
- **Complete Value** :: defines the value at the right end of the progress bar.
- **Shadow Thickness** :: the thickness of the shadow around the progress bar.
- **Show Percentage** :: a boolean value that indicates whether the percentage should be shown at all times or not.
- **Show Time** :: a boolean value that indicates whether to show the time or not.
- **Value** :: the value of the progress bar at startup.
- **Bar Type** :: either METER_BAR or METER_BOXES, defines style of progress bar.
- **# of Boxes** :: if Bar Type was defined to be METER_BOXES, defines how many boxes to divide the progress bar.
- **Top** :: Y location of the top side of the component.
- **Left** :: X location of the left side of the component.
- **Height** :: the height of the component in pixels.
- **Width** :: the width of the component in pixels.

Microline Tree

The Microline Tree component allows the developer to use a visual tree structure in an applet/application. Please reference the Microline documentation for more information.

Properties.

- **Top** :: Y location of the top side of the component.
- **Left** :: X location of the left side of the component.
- **Height** :: the height of the component in pixels.
- **Width** :: the width of the component in pixels.

******Important Notes in regard to MicroLine components******

Distribution: *When including a MicroLine third-party component, be sure to upload all of the class files in the classes/ directory to your web server. You may need to utilize codebase to tell browsers where the class files are sitting.*

Layout: *We have noticed a few quirks in Microline's third-party component behavior in relation to its position on the page. You can overcome some of this strange behavior by placing the MicroLine component into a panel, and then placing the panel where ever you wish it to go.*