CHAPTER 1

Introduction

This chapter provides an introduction to Mica, a graphics library that can be used to manage all aspects of 2D graphics applications, including user-interfaces, diagrams, graphs and animations.

Mica is named after the finely-layered, flexible, partially transparent mineral that is found in nature.

This white paper refers to Mica Version 0.90 (Alpha)

About Mica

Mica is an object-oriented graphics framework specifically crafted to support the implementation and inter-mingling of graphing editors, drawing editors, and user interfaces. To this end Mica has extensive support for display lists, event handling, action dispatching, coordinate transforms, and connectivity.

Mica is not a desktop, though desktops can be implemented on top of Mica. Mica is not a user interface toolkit, though one is included with Mica. Mica is not just a object-oriented graphics toolkit, though there are primitive graphics objects within Mica: these graphics primitives have a wealth of functionality and there are many layers of functionality on top of them. Mica is not drawing editor, though one is

included with it as a sample application. Mica is not an application framework, though Mica can be the graphics component of an application framework (see the forthcoming Cadabra white paper).

Mica is designed and written by programmers to support programmers. Whenever a part is moved or added or removed, an attribute is changed, the viewport modified, a button label changed, etc., Mica automatically updates any internal data, any layout, and the current view, if necessary. All parts derive from a richly featured base class in order that the programmer can easily add a bit more functionality to what may be historically considered a ‘simple’ part. Any part can be replaced by any other part (for example a labeled icon in a node-arc graph can be replaced by a scrolled list or a plot or an embedded internal window). Similarly, a part can be assigned to another part as an attachment without having to alter the container-part hierarchy.

Acknowledgements

We want to thank Sun Microsystems for making such a fun programming language and for leading the battle for the rest of us in the portability wars.

We would also like to acknowledge those who have written and distributed graphics toolkit source code before us: we hope you enjoy cruising this code as much as we enjoyed cruising yours; those who have published papers, manuals and books about graphics toolkits: we all rely on you to propogate the art to the rest of us; and to those whose work and ideas are held hostage by their employers: we cast Mica into the winds as yet another blow against the empire.

Naming Methodology

Every class name starts with ‘Mi’.
Every identifier name starts with ‘Mi_’.
Every interface name starts with ‘Mii’.
Every interactive event handler starts with ‘MiI’.
Every action type name ends with ‘_ACTION’.
Every event type name ends with ‘_EVENT’.
Every attribute mask lock bit ends with ‘_ATTRIBUTE_MASK_BIT’.
Every border look ends with ‘_BORDER_LOOK’.
Every write mode ends with ‘_WRITEMODE’.
Every line end style ends with ‘_LINE_END_STYLE’.
Every line style ends with ‘_LINE_STYLE’.

Every cursor ends with '_CURSOR'.
Every location ends with '_LOCATION'.
Every justification ends with '_JUSTIFIED'.

Nomenclature

1.	Action	An implementation of the MiiAction interface that is generated in response to a change in a MiPart.
2.	Action handler	An implementation of the MiiActionHandler interface that, when assigned to a MiPart, examines and optionally responds to MiiActions generated by the MiPart.
3.	Connection	A visible link between two MiParts (one of which is the *source*, the other of which is the *destination*), usually represented by a line.
4.	Connection point	The point at which a connection is attached to a MiPart.
5.	Device space	The coordinates associated with the output device. For example, pixels, if the output is the computer monitor.
6.	Draw bounds	The visible bounds of a MiPart, including shadows and attachments. The always contains the (outer) bounds of the MiPart.
7.	Enter key focus	The MiPart that has enter key focus will be the first part in it's window to be sent an MiEvent (representing the enter key) if the enter key on the keyboard is pressed by the user. This is often used by dialog and message boxes to implement their default button behavior.
8.	Event	An instance of the MiEvent class that is generated in response to the user pressing keys on the keyboard or moving/using the mouse.
9.	Event handler	An implementation of the MiiEventHandler interface which, when assigned to a MiPart, examines and optionally responds to the MiEvents received by the MiPart.
10.	Hidden	The MiPart is not viewable but still takes up screen real estate (it still has non-reversed bounds). This is often used by dialog boxes when widgets are selectively displayed/hidden (based on the state of the dialog) while the size of the dialog is to remain constant.
11.	Inner bounds	The bounds of the MiPart from the perspective of the MiPart's parts. This may be smaller than the outer bounds if the MiPart has a margin. It may also be in an entirely different coordinate system than the outer bounds (i.e. the inner bounds of a MiEditor are the world coordinates and it's outer bounds are the device coordinates).

12.	Keyboard focus	The MiPart that has keyboard focus will be the first part in it's window to be sent an MiEvent (representing the key) whenever the user presses keys on the keyboard.
13.	Line end	The start or end of a line or polyline. Line end *styles* specify what kind of arrow is to be drawn at the start or end of the line. Line end *sizes* specify the size of the arrow specified by the style.
14.	Mouse focus	The MiPart with the mouse focus is the topmost MiPart underneath the mouse pointer that accepts mouse focus.
15.	Outer bounds	The bounds of the MiPart. This is the bounds that is used by any layouts that the MiPart may be involved in.
16.	Pan	The scrolling of the contents of a MiEditor, not necessarily in just the horizontal and vertical directions.
17.	Pick	To pick a shape means to return a boolean indicating whether the MiPart intersects a given point.
18.	Pick List	A pick list is a list of MiParts (from front to back) that intersect a given point.
19.	Reversed bounds	The name for bounds (MiBounds) when they are in an uninitialized state. (The name reversed comes from the fact that bounds in this state have xmin > xmax and ymin > ymax, which are reversed).
20.	Select	The state of a MiPart when it is not deselected. The selected state is usually set when the user clicks on the MiPart, resulting in some kind of visual change to the part (handles or an outline appear around the part or the part appears *pressed*).
21.	Sensitive	The state of a MiPart when it is not insensitive. Insensitive parts usually are grayed-out and do not respond to the users actions. However, insensitive parts still receive all MiEvents (e.g. to support context sensitive help) and it is up to the part's event handlers to check the sensitivity of the part.
22.	Status bar focus	The MiPart that has status bar focus is the part that has it's st*atus bar help message* currently displayed in any status bar message field.
23.	Target List	A list of MiParts, in a MiEvent object were under the mouse cursor at the time of the event (see pick list).
24.	Target Path	A list of MiParts, in a MiEvent object, that were the MiPart directly under the mouse cursor, and all the MiParts's containers, at the time of the event.
25.	Tool hint	The smallish window that displays the *tool hint help message* of the MiPart which is currently under a paused mouse cursor.

26.	Universe space	The area within which the world space is constrained. The world space can grow and shrink (zoom) and move (pan) around in this area. This is specified by the *setUniverse(MiBounds)* method of the MiEditor class.
27.	Visible	The MiPart is viewable and has a non-reversed bounds. If the MiPart is invisible then it is not viewable and has essentially zero size.
28.	World space	The coordinate system that all MiParts contained in a MiEditor use.
29.	Zoom	An increase in magnification (zoom in) of the MiParts in a MiEditor (accomplished by reducing the size of the MiEditor's world space) or a decrease in magnification (zoom out) of the MiParts in a MiEditor (accomplished by increasing the size of the MiEditor's world space).

CHAPTER 2

Overview

This chapter presents an overview of Mica and how it works.

Design Goals

Mica is unashamedly designed for the programmer. As such the top priorities are:

1. Maximize the ease of use of the current features
2. Maximize the number (while maintaining the orthogonality) of features
3. Maximize the ease of adding features

Performance and memory size problems are tackled on a case-by-case basis if and when they arise.

Features

Written using Java (no native methods) and only a minimal amount of the AWT graphics API, Mica is extremely portable.

Many graphics objects are provided including shapes (line, rectangle, text,...), connections, widgets (push buttons, tables, tree lists, combo boxes,...), windows, dialogs and message boxes (using both native AWT frames and internal Mica frames), editors, choosers, pre-built menubars, toolbars, and graphics editors.

Subclassing from a single highly functional base class, all graphics objects therefore can be treated the same (reducing cognitive overhead), can be modified using their API or by using composition (preventing the need for of a lot of subclassing), and combined and used by other graphics objects without regard to their actual type.

The availability of the source code, for both the library and applications, makes it straight forward to mimic, copy-and-modify and debug.

A large number of behavioral objects, called event handlers, are provided which can be assigned to any graphics object. These are used, for example, by all Mica widgets to respond to the user's input. Each event handler has an event->functionality translation table which can be used to customize the precise behavior of any graphics object. The event handlers provided support, zoom, select, move, full-screen cursor, create connection, create text and much more. Special event handlers can be used to monitor and/or grab control of the event stream.

Events are Mica objects that contain useful information about the input event that generated them. All geometric information in an event is automatically transformed to the local space of each of the event handlers that examine the event. The event also contains the list of graphics objects that are lie underneath the point where the event occurred.

Actions (generated by graphics objects) are differentiated from events (generated the keyboard and mouse). Actions have four phases: request, cancel (when the request was vetoed), execute and commit.

World coordinates are used for all graphics objects for accuracy (using real numbers i.e. 'doubles') and display flexibility (magnification, birds-eye and fish-eye views, etc.). All transformations, which can be assigned to any container, are automatically used by Mica.

Any and all modifications made to any graphics object (whether to it's appearance, geometry, event handling or action handling) are automatically detected and accounted for by Mica. This includes but is not limited to updating layouts near the graphics object, updating the event and action masks of the graphics object and/or it's containers, and redrawing the graphics object.

Graphics objects are moved, resized, connected, reconnected, and animated in real-time so that the end-user does not get confused by 'disappearing graphics'.

Connections are first class graphics objects and extensive support for *having* connections is included in all graphics objects. Connections connect to graphics objects at common or any number of custom 'connection points'. Connections are automatically moved and updated by Mica. Connections are usually displayed as lines and can therefore have use of the dozen or so arrow heads and tails supplied.

Attachments are graphics objects that are assigned to other graphics objects but do not appear in the part-container hierarchy of a window. Attachments make it extremely easy to add 'resizing handles' to a selected graphics object or to add a textfield widget to a connection. Attachments can be assigned to a variety of positions with respect to their 'host' graphics object and this positioning is automatically maintained by Mica.

Any graphics object can be assigned a layout. Some layouts specify the positions of the parts inside a graphics object (i.e. a row layout), some specify the positions and connections of the parts inside a graphics object (i.e. a star graph layout), and some specify a constraint between a graphics object and another (i.e. x is to the left of y). All widgets use layouts to specify the positions of their constituent shapes.

Full support for end-user and programmatic specification of properties is provided. All text strings and icons displayed by Mica and it's applications can be changed using the plain ASCII text files: *defaults.mica* and *properties.mica*. In addition, the default widget properties and any application-specific properties can also be set in these files. Every graphics object has all 60 or so of it's attributes as properties in addition to any specific properties it may have. The translation tables of the event handlers assigned to a graphics object are also properties and in the future event handlers and widget prototype classes will also be able to be specified using properties (and property files).

Drag-and-Drop and Clipboard cut-copy-paste functionality is built-in to Mica. Any graphics object can be made a drag-and-drop source and/or target and actions for drag-over and drag-under effects are generated by the drag-and-drop manager.

Undo-redo-repeatable commands objects are used by the event handlers and a globally accessible 'transaction manager' collects these commands and manages them for programs written using Mica.

Extensive support for help is provided. Help can be assigned to any graphics object and can be a plain text string or a object that describes the text and the attributes of the text and background. The types of help are: toolhint (a smallish message), balloon (a larger toolhint with callout), statusbar (a

message to be displayed in the status bar), dialog (a message to be displayed in a dialog box). The *helpviewer* class formats and displays a very simple formatted text file as a navigable help window.

Specialized renderers can be assigned to each graphics object. Default renderers are supplied with Mica. The types of renderers are: shadow, lineEnds, border, gradient, booleanState, before, after, background, and visibility.

Architecture - The Layered Approach

Mica is layered as follows, such that the lower layers know nothing of the layers above:

- Mica Editors
- Mica Part Assemblies
- Mica Widgets
- Mica Parts, Containers and Shapes
- Mica MiRenderer, MiCanvas
- java.AWT Graphics

The AWT Layer

Mica uses the drawing capability of the AWT Graphics class, the AWT Frame and Dialog classes for window handling, and AWT Canvas for drawing output and AWT Event handling. AWT Fonts and AWT Colors are using for rendering. Upon this is built a complete user-interface toolkit combined with a 2D vector graphics library and upon *this* are application-sized widgets with which one can easily create graphical applications.

The Mica-AWT Interface Layer

The AWT Graphics class is subclassed by MiRenderer which adds an API that supports drawing in world coordinates, device coordinate specialized renderers and the pushing and popping of override attributes. The AWT Canvas class is subclassed by MiCanvas and adds support for window locking and the event handling and animation thread.

The Mica Construction Layer

All graphics (shapes, widgets, choosers, editors, windows) in Mica are MiParts, which are arranged in groups using MiContainers. Shapes (like line, circle, rectangle, text,...) are used by more sophisticated parts to create their appearance.

The Mica Widgets Layer

This layer contains standard widgets which are built using shapes and other widgets.

The Mica Parts Layer

This layer contains large assemblies of widgets into tools like choosers (font, color, line width,...) and pre-built menus, toolbars and main windows.

The Mica Editors Layer

This layer contains pre-built editors for graphing, drawing and diagramming that can be used either as stand-alone windows or incorporated in other windows.

Event and Action Handling

Simply put: Mica manages an AWT Canvas in a AWT window, drawing Mica shapes and Mica widgets in the Canvas, watching for AWT Events generated by the user in the Canvas, converting them to Mica events, forwarding these events to the shapes and widgets, who generate Mica actions that larger assemblies of widgets do something intelligent with, just like the user intended them to do.

CHAPTER 3

Parts

This chapter describes MiParts and how and why they are used. Just about everything in Mica is a MiPart and Mica has been designed in order to provide many convenient ways to display, arrange, manipulate, inquire and interact with these MiParts.

About MiParts

MiParts are the basic geometric construction element in Mica. They have a name, are drawable, have attributes, receive and process events, generate actions, and many more capabilities. Mica has been intentionally designed to have all parts be very powerful, full-featured objects. This is in order to make programming with Mica easy and rewarding (when memory considerations become paramount, lightweight and very lightweight shapes can be used).

Through the use of containers and references, a part-container hierarchy can be constructed. This event and action propagations, drawing and other aspects of this hierarchy is automatically handled

in Mica.Many traditional convolutions associated with programming GUIs are no longer required with Mica. Much of the tedious ‘housekeeping’ is handled by Mica itself, wherever possible. Examples of this are layout validation and invalidations, the enabling of actions and events, redrawing of shapes, etc.

The top levels of the MiPart Class Hierarchy

MiPart
- MiMultiPointShape
- MiConnection
- MiContainer
  - MiEditor
  - MiLayout
    - MiVisibleContainer
      - MiWidget

MiPart Functionality Overview

This section lists the major areas of functionality of every MiPart and describes the basic idea and scope of each area.

Named Resources

MiParts have an unbounded array of named resources available for you to use and methods to set, get, remove and iterate through them.

Life and Death Management

MiParts have methods to *copy(), deepCopy(), deleteSelf(), removeSelf()* (from all containers), *replaceSelf(MiPart)* which are fully aware of any Attachments and Connections the part may have.

Deep Connections

MiParts have methods which support the iteration through all connections of the part and all of it’s parts.

Drag and Drop Management

MiParts may be a source of and/or a target of a drag and drop operation. As such there are methods to indicate if such functionality is enabled (see MiAttributes), specify the drag and drop behavior, how the part will import and export data and what their valid data formats are.

Attributes

There are numerous methods to set and get individual attributes of a MiPart as well as it's assigned MiAttributes object.

Properties

Properties can be set and inquired and include all a MiPart's attributes and additional subclass-specific properties. In addition, a MiPropertyDescription can be obtained for each property that contains information about the type of the value of the property and list all valid values (if finite) and validate new values of the property.

Focus Management

Each part has the potential of having the current keyboard, mouse and/or enter-key focus. There are methods to request and inquire each kind of focus.

Select State, Sensitivity, and Visibility and Hidden State Management

There are methods to set and get the basic state of the MiPart.

Point Management

Methods to inquire, append, insert, and remove points are available for all MiParts. However, on parts that are not MiMultiPointShapes, the available points are the lower-left-hand and upper-left-hand corners and they can be inquired only.

Geometry Management

There are extensive methods to inquire and modify the geometry of every MiPart. This includes operations such as changing it's size and position. These methods are grouped into operations on the center, sides, height, width and bounds of the MiPart. In addition, basic operations such as translate, rotate and scale are available.

Pick Management

Pick management performs two functions: 1) indicating whether the MiPart intersects the given point and 2), returning a list of MiParts that intersect the given point.

Draw Management

MiParts have no draw methods that are available for your use; they are redrawn by Mica when their appearance or geometry changes. However there are methods to specify that the MiPart is to be drawn to and redrawn from a (double) buffer, to create an Image from an area of the MiPart, and to halt the current thread until the MiPart is redrawn (*waitUntilRedrawn*()). Note that a whole root window can be double buffered by using the specialized methods on their MiCanvas object.

Attachment Management

MiParts have methods to append, inquire and remove attached MiParts.

Container Content Management

All MiParts have methods to append, inquire and remove other MiParts. However these methods are only functional for MiContainers. Having MiPart implement these methods means a lot less of you having to explicitly test each MiPart to see if it is a MiContainer.

MiParts have methods that act on actual parts (*appendPart(MiPart))* and semantic parts: items (*appendItem(MiPart)*). Items are usually actual parts except in cases like MiLists (where an item is a row in the list), and like MiEditors with layers (where items are the shapes in the current layer).

Containers management

Methods are available to append, insert and inquire containers of the MiPart.

Bounds Management

Methods to set and get inner, outer and draw bounds and to set and get minimum and preferred sizes (which override those of any layout associated with the MiPart).

Invalid Area Management

Each MiPart has methods to invalidate areas within it’s bounds, causing a subsequent redraw of the MiPart. This, however, rarely if ever needs to be used because Mica automatically invalidates areas that need it.

Other methods specify whether or not the MiPart is an *opaque rectangle* (the default is that it is not unless it is an instance of MiEditor, MiTable or MiMenu). If it is a opaque rectangle then nothing is drawn underneath the MiPart. The MiPart is assigned a draw manager that takes care of this. This is useful for both speed of execution and for aesthetics of appearance.

Layout Management

Provided are the methods to set and get the MiiLayout assigned to the MiPart and to invalidate and test the validity of any such layout. The ability to invalidate the MiPart's layout, however, rarely if ever needs to be used because Mica automatically invalidates layouts that need it.

Connection Management

MiConnections can be appended, inserted, removed and inquired. Convenience methods are available to get all of a MiParts parents and children and to return whether of not the MiPart is connected to another, given, MiPart.

Connection Point Management

A MiConnectionPointManager can be assigned to the MiPart (See chapter on Connections).

Event Handling

Any number of event handlers can be assigned to any MiPart and MiParts have methods to append, insert and remove and enable/disable event handlers.

If a event handler is assigned to the MiPart and is not position dependent then it is automatically registered with the MiPart's window (if and when it has a containing window) as a global event handler (i.e. a hot key/accelerator event handler. Similarly it will be automatically removed from the window if the event handler is removed from the MiPart or if the MiPart is removed from the window).

There are also methods that inquire what events the MiPart (i.e. all of it's event handlers) is interested in.

Action Handling

A large number of methods are provided to append, insert, and remove action handlers (actually the MiiActions that are to be dispatched to the MiiActionHandler are what are registered; see the chapter on Actions). A number of methods are also available to register callbacks, which are sometimes more convenient to code that action handlers and which simply send a text String to a MiiCommandHandler object.

Action Generation

A number of actions are generated directly by the MiPart class. Some of these are generated only when there is a action handler registered that is interested in the action. These will be marked with

a *. The others will be generated and iterate through each action handler assigned to the MiPart looking for an interested handler. These others will then check a special composite handler that represents the action handlers of all of the MiPart's containers and their containers, etc. If this composite handler is interested, then the action is forwarded up the part-container hierarchy. The actions generated by the MiPart are:

- Mi_COPY_ACTION
- Mi_REPLACE_ACTION
- Mi_DELETE_ACTION
- Mi_GOT_KEYBOARD_FOCUS_ACTION
- Mi_LOST_KEYBOARD_FOCUS_ACTION
- Mi_GOT_ENTER_KEY_FOCUS_ACTION
- Mi_LOST_ENTER_KEY_FOCUS_ACTION
- Mi_GOT_MOUSE_FOCUS_ACTION
- Mi_LOST_MOUSE_FOCUS_ACTION
- Mi_SELECTED_ACTION
- Mi_DESELECTED_ACTION
- Mi_HIDDEN_ACTION
- Mi_UNHIDDEN_ACTION
- Mi_PART_VISIBLE_ACTION
- Mi_PART_INVISIBLE_ACTION
- Mi_INVISIBLE_ACTION
- Mi_VISIBLE_ACTION
- Mi_DRAW_ACTION*
- Mi_SIZE_CHANGE_ACTION*
- Mi_POSITION_CHANGE_ACTION*
- Mi_GEOMETRY_CHANGE_ACTION*
- Mi_APPEARANCE_CHANGE_ACTION*

Manipulator Management

These few methods support two kinds of manipulators: part manipulators and layout manipulators. For each of these manipulators there is a method to create the manipulator (for the MiPart or it's

layout) and a method to get the manipulator that has already been assigned to the MiPart (or it's layout) if any.

Special Containers Management

MiParts have 3 methods that return important containers of the MiPart. These methods are:

MiWindow	*getRootWindow*()
MiEditor	*getContainingEditor*()
MiWindow	*getContainingWindow*()

Debug Management

There are a number of methods that are dedicated to helping track what is happening to the MiPart. For example there is a getID() method that will a unique integer identifying the MiPart. There are also methods to iterate through event handlers and action handlers.

CHAPTER 4

Editors

This chapter describes editors and how and why they are used. Editors are used to display and interact with MiParts.

About MiEditors

MiEditors are a direct subclass of MiContainer. MiParts contained in an MiEditor are displayed using the transform as specified by the MiViewport associated to the editor.

Because an MiEditor is a subclass of MiPart, it can be assigned event handlers. In fact, there are a number of event handlers supplied with Mica that are specifically designed to be assigned to MiEditors. These event handlers provide most if not all of the functionality associated with interactive graphing and drawing editors.

About MiViewports

Every MiEditor has an associated MiViewport. MiViewports are responsible for the mapping of the coordinates of MiParts to the coordinates of the pixels on the screen. We call this the mapping the transformation of **world** coordinates into **device** coordinates.

Because this transformation defaults to having a scale factor of 1.0, the world coordinates can be considered *natural* or *virtual device* coordinates. This is because, for example, when specifying that you want a 2 pixel margin between the outside of a MiTextField widget and a MiLabel widget one just specifies that the difference in coordinates is equal to 2 world coordinates. Then, under normal circumstances, the difference will be transformed into 2 device (pixel) coordinates. Under other circumstances, the margin will be expanded or shrunk in correspondence with the current magnification. This permits the intermingling of widgets and shapes to size correctly.

The maximum and minimum world coordinates are specified by the viewport's **universe** and **miniverse**. You can use these bounds are used to put limits on magnification levels and on the size of the area the user pans around in.

The viewport should be manipulated by using methods on the MiEditor so that the MiEditor may keep its internal state and layout up-to-date. In general, however, the supplied event handlers will be sufficient to provide all desired functionality.

About Layers

The MiEditor has support for multiple layers. Each layer is a MiContainer. Layers are manipulated by using the corresponding methods in the MiEditor class.

About MiiEditorViewportSizeLayout

Because the MiEditor manages both the device and world coordinate spaces (through it's private instance of the MiViewport class) it needs to know what to do with the world bounds when the device bounds are changed. Therefore, each MiEditor has a specialized layout, which is a subclass of MiiEditorViewportSizeLayout, which manages this. The default layout is MiEditorViewportSizeIsOneToOneLayout, which keeps the sizes of the world spaces and device spaces equal.

About MiiSelectionManager

Each MiEditor has methods and an associated object that manages the selection of its MiParts. The object is a subclass of MiiSelectionManager and the default is an instance of MiSelectionManager.

CHAPTER 5

Windows

This chapter describes windows and how and why they are used. Windows are used to display and interact with MiParts and may or may not be associated with a window created by AWT.

About MiWindows

MiWindows are a direct subclass of MiEditor.

The top levels of the Window Class Hierarchy

MiPart
    MiContainer
        MiEditor

MiWindow
    MiNativeWindow
        MiNativeDialog
        MiNativeMessageDialog
    MiInternalWindow
        MiDialog
            MiMessageDialog

About MiNativeWindows

MiNativeWindows are windows that are ultimately created and managed by the underlying window system (e.g. AWT) and can be either windows, dialog boxes, or canvases. The contents of the window is always represented by a MiCanvas which is a subclass of the AWT Canvas. Windows which have an associated AWT Canvas are called **root** windows. MiNativeWindows are managed by calling methods on the MiNativeWindow class, however the associated java.awt.Frame of root windows is available (by using the getFrame() method) as is the java.awt.Canvas of any root window (by using the getCanvas() method).

About MiNativeDialogs

MiNativeDialogs are a subclass of MiNativeWindow. They are constrained to stay in front of their parent MiNativeWindow.

About MiNativeMessageDialogs

MiNativeMessageDialogs are a subclass of MiNativeDialog. They are a convenience class whose contents are automatically built from constructor arguments. MiToolkit has a number of static methods that provide even more convenient methods to create, display and wait for a user's response to a native message dialog.

About MiInternalWindows

MiInternalWindows are created and managed by Mica and can be either windows or dialog boxes. If the window has a border then the border is managed by the MiWindowBorder class.

About MiDialogs

MiDialogs are a subclass of MiInternalWindow. They are constrained to stay in from of their parent MiInternalWindow.

About MiMessageDialogs

MiMessageDialogs are a subclass of MiDialog. They are a convenience class whose contents are automatically built from constructor arguments.

About MiDragAndDropManager

MiNativeWindows have an associated instance of the MiDragAndDropManager class which is responsible for managing the pickup, drag and drop activities during a drag and drop operation.

About MiKeyboardFocusManager

Native windows have an associated instance of the MiKeyboardFocusManager class which is responsible for managing keyboard focus and enter key focus for the window. This includes interactive traversal and programmatic modification and inquiry.

About MiStatusBarFocusManager

If the window has a status bar then it will have an instance of the MiStatusBarFocusManager class which is responsible for determining what MiPart has ‘status bar focus’ and what resultant message should be displayed in the status bar message field.

About MiiKeyFocusTraversalGroup

Every window has an instance of an implementation of the MiiKeyFocusTraversalGroup which manages the order of traversal for keyboard and enter key focuses. The default implementation is of the MiLazyKeyFocusTraversalGroup class which just examines the current set of MiParts in the window and chooses the next/previous valid candidate.

CHAPTER 6

Shapes

This chapter describes shapes and how and why they are used. Shapes are used when you want to display interactive geometric shapes.

About Shapes

Shapes are the geometric building blocks of Mica. There are rectangle shapes, line shapes, text shapes and many others. Because shapes are subclasses of MiPart they can be assigned event handlers and therefore have interactive capabilities.

Some shapes, defined by a number of points (as opposed to those that are defined by a bounding box) are subclassed from **MiMultiPointShape**. The MiMultiPointShape class adds a number of methods to manipulate the defining array of points.

Because shapes are full-featured subclasses of MiParts they take up a significant amount of memory (400+ bytes). They are therefore impractical for applications that have very many shapes (fox example ECAD PC boards with 100,000+ lines). To address this shortcoming two (2) special shapes are supported: the **MiLiteShapesContainer** and **MiVeryLightweightShape**.

MiLiteShapesContainer is a shape that contains objects of type MiLightweightShape. Each of these very small MiLightweightShape objects contains only enough information to support drawing the geometry of the shape and one **tag** to allow your application to keep track of the individual MiLightweightShapes. The MiLiteShapesContainer class has methods with which you can add, remove and inquire its contents. There are a number of MiLightweightShapes provided with Mica, all of which have 'Lite' in their name (for example: MiArcLite).

MiVeryLightweightShape is for when your application needs even more memory savings and using an object per shape is impractical (for example: MiArcLite takes 24 bytes + typically the 16 bytes overhead that Java needs for house keeping = 40 bytes). MiVeryLightweightShape manages blocks of memory, typically 16K bytes in size, that are packed with only the minimal amount of information needed to define each geometric primitives. The MiVeryLightweightShape class has methods to allow adding, removing, and inquiring these geometric primitives.

Hierarchy

The definition of the shapes classes are contained in the *shapes* sub-directory. Here is the shape class hierarchy:

MiPart
- MiArc
- MiCircle
- MiEllipse
- MiEllipticalArc
- MiImage
- MiLiteShapesContainer
- MiMultiPointShape
  - MiLine
  - MiPolyline
  - MiPolyPoint
  - MiPolygon
  - MiTriangle
- MiRectangle

MiRoundRectangle
MiText
MiVeryLightweightShape

MiLightweightShape
MiArcLite
MiCircleLite
MiEllipseLite
MiImageLite
MiLineLite
MiPointLite
MiPolyLineLite
MiPolyPointLite
MiPolygonLite
MiRectLite
MiTextLite

Full featured shapes

Full featured shapes are MiParts and so have the complete set of functionality found in all MiParts. These trade off a large memory footprint and slower drawing speed for this rich set of functionality.

Lightweight shapes

Light weight shapes are Objects, not MiParts, that are grouped into containers which are instances of MiLiteShapesContainer. The MiLiteShapesContainer class is a subclass of MiPart. The lightweight shapes inside the MiLiteShapesContainer inherit the attributes of the MiLiteShapesContainer (i.e. the are all the same color,...). These trade off less functionality for a small memory footprint but retain the convenience of each shape still being an individual object.

Very lightweight shapes

Very lightweight shapes are just some raw data (i.e. ~4 doubles) in an instance of MiVeryLightweightShape. The MiVeryLightweightShape class is a subclass of MiPart. These shapes are

grouped into blocks of data (the default size is 2K). Basic attributes are specified as data in these blocks along with the shapes. These trade off minimal functionality and convenience for a minimal footprint (depending on block size and number of shapes, of course).

Rectangular Shapes

These are identified by the fact that their size can be specified by a MiBounds. Shapes of this type are manipulated by MiBoundsManipulator and created by MiICreateObject.

Multi-Point Shapes

These are identified by the fact that the must be specified by a number of points. Shapes of this type are manipulated by MiMultiPointManipulator and created by MiICreateMultiPointObject.

CHAPTER 7

Containers

This chapter describes containers and how and why they are used. You can use containers to group parts together, which can then be manipulated as a single part.

About MiContainers

A container is a type of Mica Part that you can use to contain other parts. A container is an instance of the MiContainer class which is a subclass of the MiPart class.

The parts that are added to a container are not copied into the container. Instead references are made to the parts maintained by the container. Containers may indeed contain other containers. Mica provides many methods which can be used to add, remove and inquire the parts contained in a container.

A container can be assigned a *layout* (see the chapter on layouts). These layouts can be used to arrange the parts within a container.

The Visible Container

There is a special type of container which can have a visible background and/or border. This container is the MiVisibleContainer and it is a subclass of MiLayout which is a subclass of MiContainer. It is used as the base class of all widgets in the user interface toolkit. It is also used to support the special needs of other visually compact collections of shapes. To this end the MiVisibleContainer has special methods that allow specification of both the shape of the border and how the border shrinks itself around its contents.

CHAPTER 8

Connections

This chapter describes connections and how and why they are used. Connections are used when you want to connect two MiParts together using a visible line of some sort.

About MiConnections

Connections are instances of the class MiConnection which is a subclass of MiPart. Connections connect two MiParts together, a source and a destination. Connections are visually represented by a MiPart (accessed by setGraphics() and getGraphics()) which by default is a MiLine. Connections automatically redraw themselves as their source and/or destinations move or change geometrically in some way (i.e. they are *rubberbanding* connections).

About Connection Points

Connections are attached to specific **connection points** of the source and destinations, which by default are Mi_CENTER_LOCATIONs. A number of connection points have been predefined by Mica, consisting of the 8 points of the compass and the center point. In addition, each point of the definition of the source and destination shapes can also be specified as the connection point (i.e. the 4th point of a polyline).

And finally, custom connection points can be created that can supplement or replace the other types of connection points. This is accomplished by creating and assigning a **MiConnectionPointManager** to a source and/or destination of a connection.

About MiConnectionPointManagers

MiConnectionPointManagers manage the valid connection points of one or more MiParts.

CHAPTER 9

Widgets

This chapter describes widgets and how and why they are used. Widgets are used when you want to add standard user-interface form interactors like buttons and textfields to your program.

About MiWidgets

Mica widgets are subclasses of MiWidget and MiWidget is a MiPart that is a subclass of MiVisible-Container. Widgets typically have their own event handlers to handle their interactions with the end-user; however more can be added if desired.

Because widgets are MiVisibleContainers they can have any shape. There are 8 built-in shapes:

- RECTANGLE_SHAPE
- CIRCLE_SHAPE
- ROUND_RECTANGLE_SHAPE

- DIAMOND_SHAPE
- TRIANGLE_POINTING_UP_SHAPE
- TRIANGLE_POINTING_DOWN_SHAPE
- TRIANGLE_POINTING_RIGHT_SHAPE
- TRIANGLE_POINTING_LEFT_SHAPE

which can be assigned to the widget using the *setShape*(int) method. Custom shapes can be assigned using the *setShape*(MiPart) method.

With the exception of MiTextField, widgets can contain any MiPart, which can be any other widget, any shape, text, or whatever. This allows the creation of many unique-looking widgets. Similarly, widgets can be assigned event handlers which allows the creation of unique-behaving widgets.

All widgets support the following methods to provide an easy way to set and get their primary value(s) (what the primary value(s) are is dependent, of course, on the type of the widget).

```
setValue(String)
String getValue()
setContents(Strings)
Strings getContents()
```

A widget can be specified to be one of a group of widgets whose selection state depends on the group's other widgets selection state. This is done by assigning to it the same MiRadioStateEnforcer that the other widgets have by using the *setRadioStateEnforcer*(MiRadioStateEnforcer) method.

Widget attributes

Widgets have an additional attribute bundle *for each state*. There are 6 states: normal, insensitive, selected, keyboard focus, enter-key focus and mouse focus. Whenever the widget changes state, the attributes for that state are automatically assigned to the widget. This allows the programmer to easily specify that all widgets will react visually in the same state-triggered way.

For example, the default attributes assign a hilite border to widgets what have either keyboard, enter-key or mouse focus. Individual attributes of these attribute bundles can be set using the MiToolkit class or by using specialized methods of the MiWidget class (for example *setKeyboardFocusBackgroundColor*(Color)). This feature can be disabled by using the widget's *setAutoAttributesEnabled*(boolean) method.

Widget Hierarchy

MiPart
- MiContainer
  - MiVisibleContainer
    - MiWidget
      - MiAdjuster
        - MiSlider
          - MiGauge
          - MiScrollBar
      - MiBox
      - MiLabel
        - MiButton
          - MiCheckBox
          - MiCircleToggleButton
          - MiMenuLauncherButton
            - MiOptionMenu
          - MiPushButton
          - MiSpinButton
          - MiToggleButton
        - MiMenuItem
      - MiComboBox
      - MiExpandoBox
      - MiLabeledWidget
      - MiMenu
      - MiMenuBar
      - MiOkCancelHelpButtons
      - MiPieChart
      - MiPlayerPanel
      - MiRadioBox
      - MiScrolledBox
      - MiStandardMenu
      - MiStatusBar
      - MiTabbedFolder
      - MiTable
        - MiList

MiTreeList
MiTextField
MiWindowBorder

Standard Widgets

MiLabels

Many widgets subclass the MiLabel widget. This widget displays a MiPart (which is often an instance of the MiText class) within a border. There is no interactive behavior associated with the MiLabel widget. If a string is assigned to the label (using either the constructor or the setValue(String) method) then an instance of the MiText class is automatically created. The widget's label can be retrieved for subsequent modification using the *MiPart getLabel()* method.

MiButtons

All buttons subclass the MiButton widget. The MiButton class has the capability of automatically displaying a different label for each of the following states: Normal, Selected, Insensitive and Focus (mouse and/or keyboard). This capability is activated by using the following methods (a null label assigned to a state causes the button to display the normal label for that state).

```
setNormalLabel(String)
setNormalLabel(MiPart)
MiPart getNormalLabel()
```

and similar methods for SelectedLabel, InsensitiveLabel, and FocusLabel.

MiTable

The table widget is a sophisticated widget incorporating capabilities to organize MiParts using scrollable rectangular layouts with cell by cell margin, tag, justification, sizing, and background customizations; row/column sorting, selection and moving; row and column headers and footers, cells which can span rows and/or columns, and much more. MiList and MiTreeList subclass MiTable and so MiTable's functionality is available to them as well.

The MiTable Class

One of the basic methods of the MiTable class is:

```
addCell(int rowNumber, int columnNumber, MiPart part)
```

a rowNumber is the number of a row, starting from number zero or one of the following:

- MiTable.ROW_HEADER_NUMBER
- MiTable.ROW_FOOTER_NUMBER

similarly, a columnNumber is the number of a column, starting from zero, or one of the following:

- MiTable.COLUMN_HEADER_NUMBER
- MiTable.COLUMN_FOOTER_NUMBER

Another basic method is:

```
MiTableCell getCell(int rowNumber, int columnNumber)
```

For example:

```
getCell(2, 0).getGraphics().setToolHintMessage("This is a tool hint");
```

The MiTableCell Class

Each cell in the table is represented by an instance of the MiTableCell. The MiTableCell class is responsible for the drawing, picking, tag, and cell-specific margins, justification, and sizing (there are also table-wide and row/column-wide margins, justification, and sizing options). Cells may also span multiple columns and or rows by using the MiTableCell's *setNumberOfRows*() and *setNumberOfColumns*() methods.

The MiTableCells Class

MiTableCells is a collection of instances of MiTableCell and is useful for managing rows, columns and areas of cells.

The MiTableSelectionManager Class

MiTableSelectionManager manages the selection of cells, rows, and columns and the constraints on and graphical feedback of the browsing and selection of same.

The MiGridBackgrounds Class

MiGridBackgrounds manages the backgrounds of the table on a table-wide, row, column, cell-area and cell-by-cell basis. It supports grid lines as well as arbitrary MiParts for the backgrounds.

An example

```
MiPart createScrolledTable()
{
// Create the table
table = new MiTable();

// Create the row headers (at the left of the table)
table.addCell(MiTable.ROW_HEADER_NUMBER, 0, new MiText("row 0"));
table.addCell(MiTable.ROW_HEADER_NUMBER, 1, new MiText("row 1”));

// Create the column headers (the top of the table)
table.addCell(0, MiTable.COLUMN_HEADER_NUMBER, new MiText("column 0"));
table.addCell(1, MiTable.COLUMN_HEADER_NUMBER, new MiText("column 1"));

// Create the column footers (at the bottom of the table)
table.addCell(0, MiTable.COLUMN_FOOTER_NUMBER, new MiText("column 0"));
table.addCell(1, MiTable.COLUMN_FOOTER_NUMBER, new MiText("column 1"));

// Create an array of cells 3 rows and 2 columns in size
table.addCell(0, 0, new MiText("cell 0,0"));
table.addCell(0, 1, new MiText("cell 0,1"));
table.addCell(1, 0, new MiText("cell 1,0"));
table.addCell(1, 1, new MiText("cell 1,1"));
table.addCell(2, 0, new MiText("cell 2,0"));
table.addCell(2, 1, new MiText("cell 2,1"));

// Specify the minimum vertical size
table.setMinimumNumberOfVisibleRows(2);

// Specify the preferrred vertical size
table.setPreferredNumberOfVisibleRows(2);

// Specify when to add a vertical scrollbar
table.setMaximumNumberOfVisibleRows(2);

// Specify a tool hint for the cell in the upper left hand corner
table.getCell(0, 0).setToolHintMessage("This is cell 0,0”);

// Specify a special mouse cursor for the upper left hand corner
table.getCell(0, 0).setContextCursor(Mi_HAND_CURSOR);
```

```
// Create a raised, beveled rectangle to use for each of the table's cells
MiRectangle rect = new MiRectangle();
rect.setBorderLook(Mi_RAISED_BORDER_LOOK);
rect.setBackgroundColor(getBackgroundColor());
table.getBackgroundManager().appendCellBackgrounds(rect);

// Specify that the 1th column can resize horizontally in order the
// entire center of the table will be occupied horizontally
table.getMadeColumnDefaults(1).setColumnHorizontalSizing(Mi_EXPAND_TO_FILL);

// Make the table scrollable by putting it inside a scrolled box
MiScrolledBox scrolledBox = new MiScrolledBox(table);

// Specify that the scrolledBox fades in to whatever container it may have
scrolledBox.setBackgroundColor(Mi_TRANSPARENT_COLOR);
// Return the scrollable table
return(scrolledBox);
}
```

MiTreeList

The MiTreeList class subclasses the MiTable class. It adds a large number of methods to provide a convenient API for the programmer. In particular there are methods that directly support the use of a *tag* (i.e. user info) for each row in the tree list. This is most useful for the programmer who needs to know what each row represents.

CHAPTER 10

Attributes

This chapter describes attributes and how and why they are used. Attributes are used when you want to customize the appearance or behavior of a MiPart in a manner that Mica has been written to support (for example to set the color of a MiPart to red).

About MiAttributes

Every MiPart has an associated collection of attributes, which are stored in an instance of the MiAttributes class. This collection contains attributes representing approximately 60 different aspects of a MiPart's appearance and behavior.

Some major attributes are:

- Color
- BackgroundColor

- BackgroundImage
- LineWidth
- Font
- BorderLook
- ContextMenu
- ContextCursor
- ToolHintMessage
- BalloonHelpMessage
- StatusHelpMessage
- DialogHelpMessage

About Attribute Management

Instances of the MiAttributes class can be explicitly **shared** among any number of MiParts, but usually, each MiPart has it's own private instance (however, instances of MiAttributes are cached and shared internally to Mica to increase both time and memory efficiency). Whenever, for example, the setColor() method is called on a MiPart, a new MiAttributes is created (or reused if found within the MiAttribute cache) consisting of the previous MiAttributes's attribute values merged with the new value of the color attribute.

Any attribute in an instance of a MiAttributes class can have it's value be **inherited** from another MiAttributes. This inheritance is specified on a attribute by attribute basis (by the MiAttributes.setIsInheritedAttribute() method). Values are inherited from the attribute's associated MiPart's container. Additionally, if an attribute is set to a specific value then it will no longer inherit values. This allows you to *override* any inherited value (e.g. the programmatically specified value takes precedence over any inherited value).

Widget attribute management has more functionality and you are urged to peruse that chapter as well.

The Attribute Methods

Every MiPart has a large number of attributes. The value of these attributes can be modified and inquired by either using specialized methods of the MiPart, by assigning an MiAttributes instance

to the MiPart, or by using one of the general methods of the MiPart. For example, to set the color of a MiPart p, one can do either

```
p.setColor(MiColorManager.blue)
p.setAttributeValue(Mi_COLOR_ATT_NAME, MiColorManager.blue)
p.setAttributes(p.getAttributes().setColor(MiColorManager.blue))
```

The MiPart general methods are:

```
void      setAttributeValue(String name, Object value)
void      setAttributeValue(String name, int value)
void      setAttributeValue(String name, double value)
void      setAttributeValue(String name, boolean value)
void      setAttributeValue(String name, String value)
String    getAttributeValue(String name)
boolean   hasAttribute(String name)
```

See Appendix A for a detailed list of Attributes.

MiPushAttributes and MiPopAttributes

The MiPushAttributes part specifies what attributes are to be used when drawing any MiParts which follow the MiPushAttributes part and come before a MiPopAttributes part or until the last part in the root window is drawn. The attributes of a MiPushAttributes can be considered to ‘override’ all attributes of succeeding parts.

CHAPTER 11

Layouts

This chapter describes layouts and how and why they are used. You can use layouts to specify how parts are arranged within containers. There are many layouts provided with Mica. Some support graphs (nodes and their connections). Others support arrangements of a container's parts irrespective of any connections the parts may have.

About MiiLayouts

The layout interface, MiiLayout, has methods that, like the AWT Layout class, support the automatic layout of associated parts. In addition, it has methods that allow the manipulation of the layout by specialized editors.

The primary layout object is MiLayout. This layout object is a subclass of MiContainer so it can (but does not usually) contain the parts it is to layout (see Caveats).

Shape Layouts

Shape layouts position shapes without regard to any connections they may have (to each other or to external shapes). These shape layouts are most often used to organize shapes together into rectangular containers. The layouts supplied with Mica are:

- MiColumnLayout
- MiGridLayout
- MiRowLayout

These layouts have many sizing, justification and margin options that allow you to easily specify that layout of most, if not all, of the situations you will encounter. The sizing options that Mica supports are:

- Mi_SAME_SIZE
- Mi_EXPAND_TO_FILL

The horizontal justification options are:

- Mi_LEFT_JUSTIFIED
- Mi_RIGHT_JUSTIFIED
- Mi_JUSTIFIED
- Mi_CENTER_JUSTIFIED

The vertical justification options are:

- Mi_BOTTOM_JUSTIFIED
- Mi_TOP_JUSTIFIED
- Mi_JUSTIFIED
- Mi_CENTER_JUSTIFIED

The margins that can be specified are:

- Inset — The margins between the shapes as a group and their container's inner boundry.
- Alley — The horizontal/vertical distance between shapes.
- Cell — The margin around each shape.

Graph Layouts

Graph layouts position shapes with respect to the connections the shapes have to each other. These graph layouts are most often used to organize shapes into *node-arc* (icons and lines) graphs. The layouts supplied with Mica are:

- Mi2DMeshGraphLayout
- MiCrossBarGraphLayout
- MiLineGraphLayout
- MiOmegaGraphLayout
- MiOutlineGraphLayout
- MiRingGraphLayout
- MiStarGraphLayout
- MiTreeGraphLayout
- MiUndirGraphLayout

Special Layouts

MiPolyLayout

The MiPolyLayout allows the assignment of multiple layouts to one MiContainer.

MiPolyConstraint

The MiPolyConstraint layout allows multiple constraints can be specified using the MiRelativeLocationConstraint class. Using this constraint class, many constraints can be specified at one time by adding them to a single instance of the MiPolyConstraint class. The MiRelativeLocationConstraint class supports many kinds of two-party (master-slave) constraints:

- LEFT_OF
- RIGHT_OF
- TOP_OF
- BOTTOM_OF
- INSIDE_LEFT_OF
- INSIDE_RIGHT_OF
- INSIDE_TOP_OF

- INSIDE_BOTTOM_OF
- CENTER_OF
- INSIDE_OF
- OUTSIDE_OF
- SAME_ROW_AS
- SAME_COLUMN_AS
- SAME_WIDTH_AS
- SAME_HEIGHT_AS
- SAME_WIDTH_AS_PRESERVE_ASPECT
- SAME_HEIGHT_AS_PRESERVE_ASPECT
- SAME_SIZE_AS
- SAME_SW_INSIDE_CORNER
- SAME_SE_INSIDE_CORNER
- SAME_NW_INSIDE_CORNER
- SAME_NE_INSIDE_CORNER
- SAME_SW_OUTSIDE_CORNER
- SAME_SE_OUTSIDE_CORNER
- SAME_NW_OUTSIDE_CORNER
- SAME_NE_OUTSIDE_CORNER
- INSIDE_LEFT_CENTER_OF
- INSIDE_RIGHT_CENTER_OF
- INSIDE_TOP_CENTER_OF
- INSIDE_BOTTOM_CENTER_OF

Manipulating Layouts

Most layouts (all except MiPolyLayout) can be *edited* visually by the end-user, if so desired. If the layout is created with the *manipulatable* argument set to true, then:

- The layout can be initialized to have a minimum number of ‘nodes’ and the connections necessary to maintain the topology of the layout, if any. These ‘nodes’ are instances of MiPlaceHolder that can be replaced by dragged and dropped upon.

- Nodes can be added to the layout by dragging and dropping the nodes on the layout.
- When the layout is selected by the end-user, it is assigned a ‘layout manipulator’ This layout manipulator can be used to insert or append more place holders, to delete place holders and nodes, to move the selection point (cursor) around the layout (using the cursor keys and page up, page down, home and end keys).

CHAPTER 12

Events

This chapter describes events and how and why they are used. You can use events and event handlers to add event-specific behavior to any part.

About MiEvents

Events are generated by Mica in response to the user moving the mouse, using the mouse buttons or using the keyboard. In addition there is a timer event generated every second and an idle event generated when no event has occurred for a specified amount of time.

Events are objects that are instances of the MiEvent class. They contain a significant amount of information about the state of the system at the time of the event. This information includes the location of the mouse (in both device and world coordinates), the list of parts underneath the mouse, the parentage of the part immediately under the mouse, the time of the event and more.

About MiiEventHandlers

Event handlers are objects that implement the MiiEventHandler interface. Event handlers can be assigned to any MiPart. The MiEventHandler class is supplied with Mica and provides a number of conveniences. For example, you can specify what events the event handler is interested in.

Event handlers all implement a method called *processEvent*(). In this method you can perform some computations and then optionally **absorb** the event to prevent anyone else from seeing the event.

There are three (3) categories of event handlers that Mica recognizes. In addition, any event handler can **grab** all events before they are forwarded to any other event handler (after these events have been forwarded to all MiEventMonitors).

1. **MiEventMonitors** receive all events they are interested in at all times. MiEventMonitors cannot absorb events. This type of event handler is often used to implement things like the display of the current mouse x, y location, which needs to be continuously updated, regardless of the current operation the user may be involved in.
2. **MiShortCutHandlers** receive all events they are interested in that occur within the window of the MiPart they are assigned to. This type of event handler is often used to implement things like the 'accelerators' associated with buttons or menu items, which need to be automatically added, removed or desensitized whenever the associated button or many items is added, removed or desensitized.
3. **MiEventHandlers** receive all events they are interested that (optionally) occur within the bounds of the MiPart they are assigned to.

Using MiiEventHandlers

Typically event handlers are assigned to MiParts to add 'feel' to a part. For example:

```
editor.appendEventHandler(new MiIZoomAroundMouse());
```

This adds a 'feel' such that when the user clicks the middle (shift+right) button of the mouse, the contents of the editor are magnified (de-magnified). The mapping of user events to functionality in most event handlers is controlled by a easily accessible translation table.

MiParts has the following methods in support of MiiEventHandlers:

```
appendEventHandler(MiiEventHandler)
```

```
insertEventHandler(MiiEventHandler, int)
removeElementHandler(MiiEventHandler)
int getNumberOfEventHandlers()
MiiEventHandler getEventHandler(int)
MiiEventHandler getEventHandler(String)
MiiEventHandler getEventHandlerWithClass(String)
MiEvent[] getLocallyRequestedEventTypes()
setEventHandlingEnabled(boolean)
boolean getEventHandlingEnabled()
int dispatchEvent(MiEvent)
```

Mica-supplied MiiEventHandlers

These are the event handlers that are supplied with Mica. The trigger events associated with any of these event handlers can be changed programmatically by using the MiEventHandler method:

```
setEventToCommandTranslation(String commandToGenerate, MiEvent event)
```

- **MiIClickAndDrop** - this event handler, when assigned to an MiEditor, creates a ‘stamp’-like behavior (i.e. each click of the left mouse button causes the creation of a associated shape at the current mouse location).
- **MiICreateConnection** - this event handler, when assigned to an MiEditor, provides the end-user the capability of interactively connecting shapes together with the mouse.
- **MiICreateMultiPointObject** - this event handler, when assigned to an MiEditor, provides the end-user the capability of interactively creating multi-point shapes (for example lines and polylines) with the mouse.
- **MiICreateObject** - this event handler, when assigned to an MiEditor, provides the end-user the capability of interactively creating shapes defined only by their size (for example rectangles and ovals) with the mouse.
- **MiICreateText** - this event handler, when assigned to an MiEditor, provides the end-user the capability of interactively creating text with the mouse and keyboard.
- **MiIDeleteObjectUnderMouse** -
- **MiIDeleteSelectedObjects**
- **MiIDeselectAll**
- **MiIDisplayContextCursor**
- **MiIDisplayContextMenu**

- **MiIDisplayHelpDialog**
- **MiIDisplayToolHints**
- **MiIDragAndCopyWithMouse**
- **MiIDragBackgroundPan**
- **MiIDragObjectUnderMouse**
- **MiIDragSelectedObjects**
- **MiIDragger**
- **MiIExecuteActionHandler**
- **MiIExecuteCommand**
- **MiIFlowEditorEventHandler**
- **MiIFullScreenCurso**
- **MiIJumpPan**
- **MiIMouseEnterAndExit**
- **MiIMouseFocus**
- **MiINormalizedPan**
- **MiIOnePtPan**
- **MiIPan**
- **MiIPartInspector**
- **MiIPopup**
- **MiISetDebugTraceModes**
- **MiIPrintGraphicsStructures**
- **MiIPrintPostScript**
- **MiIReCalcLayouts**
- **MiIRedraw**
- **MiIRubberbandBounds**
- **MiIRubberbandPoint**
- **MiISelectArea**
- **MiISelectObjectUnderMouse**
- **MiIZoomArea**
- **MiIZoomAroundMouse**
- **MiPlayEventSound**

CHAPTER 13

Actions

This chapter describes actions and how and why they are used. You can use actions and action handlers to add action-specific behavior to any MiPart.

About MiiActions

Actions are generated by Mica in response to the changes in a MiPart. These are changes like *selection*, *deletion*, and *movement*. In addition many specialized shapes, like widgets, have specialized actions that they generate (for example *scrolled*).

Actions are objects that are implementations of the MiiAction class. They contain a significant amount of information about the state of the MiPart at the time of the action. This information includes the type of the action, user information, MiPart-specific information and a user-specified named property list.

There are four (4) different **phases** of actions that Mica generates.

- An action is generated with the **Request** phase to give any registered action handlers the opportunity to veto the action
- An action is generated with the **Cancel** phase to inform any registered action handlers that the action was vetoed
- An action is generated with the **Execute** phase to allow any registered action handler to actually execute the change the action represents
- An action is generated with the **Commit** phase to inform any registered action handlers that the change the action represents has occurred

Actions will be absorbed in the Request, Execute and Commit phases if an action handler returns False. Only some actions will have an Execute phase. The phase can be inquired by using the action's isPhase(int phase) method or by using:

```
isRequestPhase()
isCancelPhase()
isCommitPhase()
isExecutePhase()
```

About MiiActionHandlers

Action handlers are objects that implement the MiiActionHandler interface. Action handlers can be assigned to any MiPart. This is accomplished by registering the corresponding MiiAction with the MiPart. Whenever an MiiAction is dispatched, it is these registered actions that are passed to their corresponding MiiActionHandlers.

Using MiiActions and MiiActionHandlers

MiiActionHandlers have only one required method:

```
boolean     processAction(MiiAction action)
```

This method can inquire (if necessary) the MiiAction to see what action actually occurred and then preform any functionality desired. Subsequently it will return True, if it is OK that other MiiAction-Handlers see this action, or False, if not (i.e. absorbs the MiiAction).

Actions are only dispatched if they have been registered with a MiPart (as noted above: MiiActions are registered with MiParts, and MiiActionHandlers are assigned to MiiActions). There are many convenience methods available to register MiiActions. The simplest of which is:

```
void    MiPart.appendActionHandler(MiiAction action)
```

MiiActionHandlers are typically assigned to MiiActions in the MiiAction's constructor:

```
public MiAction(MiiActionHandler handler, int validActionType)
```

The type of the action, specified using the *validActionType* parameter, can be one of many available action types (which are specified in the file: MiiActionTypes.java). An example of a type would be:

```
MiiActionTypes.Mi_SELECTED_ACTION
```

In addition, the type parameter can include information about the phase of the desired action. An example might be:

```
MiiActionTypes.Mi_SELECTED_ACTION
        + MiiActionTypes.Mi_REQUEST_ACTION_PHASE
```

In addition, the type parameter can include information about whether actions that occur in the parts of the MiPart that the MiiAction is registered with are desired. The options available for this are:

```
MiiActionTypes.Mi_ACTIONS_OF_PARTS_OF_OBSERVED
```

This will cause the actions ONLY of the parts of the MiPart to be dispatched to the registered action handler.

```
MiiActionTypes.Mi_ACTIONS_OF_OBSERVED
```

This will cause the actions ONLY of the MiPart itself to be dispatched to the registered action handler. This is the default.

```
MiiActionTypes.Mi_ACTIONS_OF_PARTS_OF_OBSERVED
        + MiiActionTypes.Mi_ACTIONS_OF_OBSERVED
```

This will cause the actions of BOTH the parts of the MiPart AND of the MiPart itself to be dispatched to the registered action handler.

Note that during registration and de-registration of MiiActions with MiParts, Mica automatically enables and disables the propagation of MiiActions. In this way no time or aesthetic penalty is incurred by your code because of the need to explicitly enable and disable each specific action.

Examples of Using MiiActions and MiiActionHandlers

For example, if you wanted to be notified whenever the user drags and drops something into your MiEditor you might have:

```
class myDrawEditor extends MiEditorWindow implements MiiActionHandler
        {
        public                  myDrawEditor()
```

```
        {
        super("myDrawEditor", new MiBounds(0.0, 0.0, 500.0, 500.0));
        buildEditorWindow();
        getEditor.setIsDragAndDropTarget(true);
        getEditor().appendActionHandler(new MiAction(this,
                                Mi_DATA_IMPORT_ACTION);
        }
public  boolean         processAction(MiiAction action)
        {
        if (action.hasActionType(Mi_DATA_IMPORT_ACTION))
                {
                // Your code here...
                }
        return(true);
        }
}
```

If you also wanted to be notified when something is dropped on top of MiParts in the editor then you would add the following:

```
class myDrawEditor extends MiEditorWindow implements MiiActionHandler
        {
        public                  myDrawEditor()
        {
        super("myDrawEditor", new MiBounds(0.0, 0.0, 500.0, 500.0));
        buildEditorWindow();
        getEditor.setIsDragAndDropTarget(true);
        getEditor().appendActionHandler(new MiAction(this,
                        Mi_DATA_IMPORT_ACTION);
        getEditor().appendActionHandler(new MiAction(this,
                        Mi_DATA_IMPORT_ACTION
                        + Mi_ACTIONS_OF_PARTS_OF_OBSERVED));
        }
public  boolean         processAction(MiiAction action)
        {
        if (action.hasActionType(Mi_DATA_IMPORT_ACTION))
                {
                // Your code here...
                }
        else if (action.hasActionType(Mi_DATA_IMPORT_ACTION
                + Mi_ACTIONS_OF_PARTS_OF_OBSERVED))
```

```
                {
                // Your code here...
                }
        return(true);
        }
}
```

Then if you also wanted to allow drag and dropping on only some of the MiParts in the editor then you could assure that they have *isDragAndDropTarget*() equal to false, or if whether the parts are valid targets depends on what is being dropped on them, the one would add the following:

```
class myDrawEditor extends MiEditorWindow implements MiiActionHandler
        {
        public                  myDrawEditor()
                {
                super("myDrawEditor", new MiBounds(0.0, 0.0, 500.0, 500.0));
                buildEditorWindow();
                getEditor.setIsDragAndDropTarget(true);
                getEditor().appendActionHandler(new MiAction(this,
                                Mi_DATA_IMPORT_ACTION,
                                Mi_DATA_IMPORT_ACTION
                                        + Mi_ACTIONS_OF_PARTS_OF_OBSERVED,
                                Mi_DATA_IMPORT_ACTION
                                        + Mi_ACTIONS_OF_PARTS_OF_OBSERVED
                                        + Mi_REQUEST_ACTION_PHASE));
                }
        public  boolean         processAction(MiiAction action)
                {
                if (action.hasActionType(Mi_DATA_IMPORT_ACTION))
                        {
                        // Your code here...
                        }
                else if (action.hasActionType(Mi_DATA_IMPORT_ACTION
                        + Mi_ACTIONS_OF_PARTS_OF_OBSERVED))
                        {
                        // Your code here...
                        }
                else if (action.hasActionType(Mi_DATA_IMPORT_ACTION
                        + Mi_ACTIONS_OF_PARTS_OF_OBSERVED
                                        + Mi_REQUEST_ACTION_PHASE))
                        {
```

```
                    MiDataTransferOperation transfer
                        = (MiDataTransferOperation )action.getActionSystem-
Info();
                    MiPart obj = (MiPart )transfer.getSource();
                    MiPart target = transfer.getTarget();
                    // boolean valid;
                    // Your code here to check validity of obj dropping on target
                    // if (!valid)
                        action.veto()
                    }

                return(true);
                }
        }
```

Methods to Assign Actions/Handlers to MiParts

MiiAction Methods

appendActionHandler(MiiAction action)
insertActionHandler(MiiAction action, int index)
removeActionHandler(MiiAction action)

MiiActionHandler Methods

removeActionHandlers(MiiActionHandler handler)
appendActionHandler(MiiActionHandler handler, int validAction)
appendActionHandler(MiiActionHandler handler, int validAction1, int validAction2)

MiiCommandHandler Methods

appendCallback(MiiCommandHandler command, String argument, MiEvent event)
appendCallback(MiiCommandHandler command, String argument, int validActions)
appendCallback(MiiCommandHandler command, String argument)
removeCallback(MiiCommandHandler command)
removeCallback(MiiCommandHandler command, String argument)

MiEvent Methods

insertActionHandler(MiiAction action, MiEvent event, int index)
appendActionHandler(MiiAction action, MiEvent event)

Action Types

These are found in MiiActionTypes.java.

Mi_CREATE_ACTION
Mi_DELETE_ACTION
Mi_COPY_ACTION
Mi_REPLACE_ACTION
Mi_REPLACE_PARENT_ACTION

Mi_DRAG_AND_DROP_PICKUP_ACTION
Mi_DRAG_AND_DROP_MOVE_ACTION
Mi_DRAG_AND_DROP_ENTER_ACTION
Mi_DRAG_AND_DROP_EXIT_ACTION
Mi_DRAG_AND_DROP_PAUSE_ACTION
Mi_DRAG_AND_DROP_CONTINUE_ACTION
Mi_DRAG_AND_DROP_CANCEL_ACTION
Mi_DRAG_AND_DROP_COMMIT_ACTION

Mi_SELECTED_ACTION
Mi_DESELECTED_ACTION
Mi_ACTIVATED_ACTION
Mi_SELECT_REPEATED_ACTION
Mi_GOT_MOUSE_FOCUS_ACTION
Mi_LOST_MOUSE_FOCUS_ACTION
Mi_GOT_KEYBOARD_FOCUS_ACTION
Mi_LOST_KEYBOARD_FOCUS_ACTION
Mi_GOT_ENTER_KEY_FOCUS_ACTION
Mi_LOST_ENTER_KEY_FOCUS_ACTION
Mi_INVISIBLE_ACTION
Mi_VISIBLE_ACTION
Mi_PART_VISIBLE_ACTION

Mi_PART_INVISIBLE_ACTION
Mi_HIDDEN_ACTION
Mi_UNHIDDEN_ACTION
Mi_TEXT_CHANGE_ACTION
Mi_MENU_POPPED_UP_ACTION
Mi_MENU_POPPED_DOWN_ACTION
Mi_TABBED_FOLDER_OPENED_ACTION
Mi_TABBED_FOLDER_CLOSED_ACTION
Mi_INVALID_VALUE_ACTION
Mi_VALUE_CHANGED_ACTION
Mi_ENTER_KEY_ACTION
Mi_NODE_EXPANDED_ACTION
Mi_NODE_COLLAPSED_ACTION

Mi_ITEM_SELECTED_ACTION
Mi_ITEM_DESELECTED_ACTION
Mi_ITEM_BROWSED_ACTION
Mi_ITEM_DEBROWSED_ACTION
Mi_ITEM_ADDED_ACTION
Mi_ITEM_REMOVED_ACTION
Mi_ALL_ITEMS_SELECTED_ACTION
Mi_ALL_ITEMS_DESELECTED_ACTION
Mi_NO_ITEMS_SELECTED_ACTION
Mi_ONE_ITEM_SELECTED_ACTION
Mi_MANY_ITEMS_SELECTED_ACTION
Mi_ITEM_SCROLLED_ACTION
Mi_ITEMS_SCROLLED_AND_MAGNIFIED_ACTION

Mi_EDITOR_VIEWPORT_CHANGED_ACTION
Mi_EDITOR_WORLD_TRANSLATED_ACTION
Mi_EDITOR_WORLD_RESIZED_ACTION
Mi_EDITOR_DEVICE_TRANSLATED_ACTION
Mi_EDITOR_DEVICE_RESIZED_ACTION
Mi_EDITOR_UNIVERSE_RESIZED_ACTION
Mi_EDITOR_CONTENTS_GEOMETRY_CHANGED_ACTION

Mi_WINDOW_CLOSE_ACTION
Mi_WINDOW_ICONIFY_ACTION
Mi_WINDOW_DEICONIFY_ACTION
Mi_WINDOW_OPEN_ACTION
Mi_WINDOW_OK_ACTION
Mi_WINDOW_CANCEL_ACTION
Mi_WINDOW_FULLSCREEN_ACTION
Mi_WINDOW_NORMALSIZE_ACTION

Mi_CLIPBOARD_NOW_HAS_DATA_ACTION
Mi_TRANSACTION_MANAGER_CHANGED_ACTION
Mi_DATA_IMPORT_ACTION
Mi_CONNECTION_SOURCE_ACTION
Mi_CONNECTION_DESTINATION_ACTION
Mi_CONNECTED_ACTION
Mi_STATUS_BAR_FOCUS_CHANGED_ACTION
Mi_ICONIFY_ACTION
Mi_DEICONIFY_ACTION
Mi_GROUP_ACTION
Mi_UNGROUP_ACTION

Mi_GEOMETRY_CHANGE_ACTION
Mi_SIZE_CHANGE_ACTION
Mi_POSITION_CHANGE_ACTION
Mi_APPEARANCE_CHANGE_ACTION
Mi_DRAW_ACTION

About MiActionManager

This class is a globally accessible (i.e. it is a singleton) API to the action registry. All possible action types and their names are registered here.

The MiActionManager allows components outside of Mica to generate actions that look and behave just like Mica's built-in actions. This is accomplished by the component registering it's

unique action names with this manager and getting back unique action type values. These values can be used anywhere just like Mica action types.

For example: the Mica-supplied MiPlayerPanel class contains the following lines which allow the users of the class to access and use it's action type Mi_PLAYER_PANEL_ACTION just like a Mica built-in action types:

```
public static final String Mi_PLAYER_PANEL_ACTION_NAME
        = "playerPanelStateChange";
public static final int Mi_PLAYER_PANEL_ACTION
        = MiActionManager.registerAction(Mi_PLAYER_PANEL_ACTION_NAME);
```

CHAPTER 14

Part Assemblies

This chapter describes part assemblies and how and why they are used. You can use part assemblies to easily and quickly build entire graphical applications.

MiEditorWindow

The MiEditorWindow is a complete, customizable main window for a typical graphics application. It is a subclass of MiNativeWindow. Using the buildEditorWindow() method, one creates the contents of the window including a MiEditorMenuBar, MiEditorToolBar, MiEditorStatusBar, MiEditorPalette and, of course, a scrollable MiEditor.

MiiCommandManager

The MiiCommandManager interface (and it's MiCommandHandler implementation) support the registration of MiWidget-command pairs. The MiEditorMenuBar and MiEditorToolBar take a MiiCommandManager as an argument and register all of their widgets and their commands with the MiiCommandManager. Since MiEditorWindow implements MiiCommandManager it just passes in itself as an argument to these classes. The MiiCommandManager interface (and MiEditorWindow) support the following methods:

```
setCommandAvailability(String command, boolean flag)
setCommandAvailability(String command, boolean flag, String statusHelpMsg)
setCommandState(String command, boolean flag)
setCommandState(String command, String state)
setCommandLabel(String command, String label)
setCommandOptions(String command, Strings options)
```

These methods set the sensitivity, the state of a boolean widget, the current value of a multi-valued widget, the label of a button-like of menuitem-like widget, or the values of a multi-valued widget. For example:

```
myEditorWindow.setCommandAvailability(Mi_SAVE_COMMAND_NAME, false);
```

will cause both the 'Save' menuitem and toolbar button to be insensitive (grayed-out).

Menubars

Mica-supplied menubars are similar to Mica toolbars (See section on Toolbars below). There is a base functionality and then there is an implementation using that functionality that provides the most common features that can also be programmatically added to or removed from. The base functionality is provided by the MiMenubar class. The supplied implementation is provided by a quite of pulldown menus that are supported by the MiEditorMenuBar class.

The Class Hierarchy

```
MiMenuBar
        MiEditorMenuBar
MiiContextMenu
        MiEditorMenu
                MiConnectMenu
                MiEditMenu
```

MiFileMenu
MiFormatMenu
MiGraphMenu
MiHelpMenu
MiLayoutMenu
MiShapeMenu
MiToolsMenu
MiViewMenu

MiCommand
MiCommandWidgetCommand
MiConnectMenuCommands
MiEditMenuCommands
MiFileMenuCommands
MiFormatMenuCommands
MiGraphMenuCommands
MiHelpMenuCommands
MiLayoutMenuCommands
MiShapeMenuCommands
MiToolsMenuCommands
MiViewMenuCommands

The EditorMenuBar

The supplied implementation provides support for a menubar with the features frequently found in graphics editor applications. This includes a number of standard pulldown menus and their associated functionality. This menubar can be customized by adding and/or removing pulldown menus and pulldown menu options.

This menubar is created by the *MiEditorMenuBar* class. It includes the standard File, Edit, View, Shape, Connect, Format and Help pulldown menus. All standard accelerators and mnemonic keys are automatically supported. The pulldowns include the following functionality:

File

- New...
- Open...
- Save
- Save As...
- Close

- Print Setup...
- Print
- Quit

Edit

This menubar pulldown is activated by clicking on 'Edit' in the menubar or by using the short cut key Meta-E.

- Undo Activated with mouse or short cut key Ctrl-W (or key 'u' if menu is visible). Causes the undoing of the last editing operation. This item is disabled and dimmed if there is nothing to undo.
- Redo Activated with mouse or short cut key Ctrl-Z (or key 'r' if menu is visible). Causes the undoing of the last editing operation. This item is disabled and dimmed if there is nothing to redo, if nothing has been undone.
- Cut Activated with mouse or short cut key Ctrl-Y (or key 't' if menu is visible). Causes the removal of any selected items in the editor area, moving them to the clipboard. This item is disabled and dimmed if there is nothing selected in the editor.
- Copy Activated with mouse or short cut key Ctrl-C (or key 'c' if menu is visible). Causes the copying of any selected items in the editor area, moving them to the clipboard. This item is disabled and dimmed if there is nothing selected in the editor.
- Paste Activated with mouse or short cut key Ctrl-V (or key 'p' if menu is visible). Causes the copying of any items in the clipboard to the center of the current editor area. This item is disabled and dimmed if there is something selected in the editor or if the clipboard is empty.
- Delete Activated with mouse or short cut key <delete> (or key 'd' if menu is visible). Causes the deletion of any selected items in the editor area. This item is disabled and dimmed if there is nothing selected in the editor.
- Select All Activated with mouse (or key 's' if menu is visible). Causes the selection of all items in the editor.
- Deselect All Activated with mouse or short cut key <Esc> (or key 'a' if menu is visible). Causes the de-selection of any selected items in the editor. This item is disabled and dimmed if there is nothing selected in the editor.
- Duplicate Activated with mouse (or key 'l' if menu is visible). Causes the copying of any selected items in the editor. The copies are placed next to their copied items. This item is disabled and dimmed if there is nothing selected in the editor.

View

This menubar pulldown is activated by clicking on 'View' in the menubar or by using the short cut key Meta-V.

- Zoom In Activated by selecting menu item or by clicking the middle mouse button (with or without the shift or control key held down) inside the graphics editor. Selecting the menu item causes the zoom in to be centered around the middle of the graphics editor. Clicking the middle mouse button inside the graphics editor causes the zoom in to be centered around the current mouse position.
- Zoom Out Activated by selecting menu item or by clicking the right mouse button (with the shift or control key held down) inside the graphics editor. Selecting the menu item causes the zoom in to be centered around the middle of the graphics editor. Clicking the middle mouse button inside the graphics editor causes the zoom in to be centered around the current mouse position.
- View All Activated with mouse or short cut key Ctrl-W (or key 'a' if menu is visible). Causes the editor to 'zoom all of the way out' and the scrollbars to disappear (i.e. sets the magnification to the smallest possible value).
- View Previous Activated with mouse or short cut key Ctrl-R (or key 'p' if the menu is visible). Causes the view in the editor to return to the previous location and magnification level. This item is disabled and dimmed if there isn't a previous view.
- View Next Activated with mouse or short cut key Ctrl-T (or key 'n' if the menu is visible). Causes the view in the editor to advance to the next view. This item is disabled and dimmed if there is no next view (i.e. 'view previous' has not been used).
- Redraw Activated with mouse or short cut key Ctrl+L ((or key 'r' if the menu is visible).
- Toolbar Activated with mouse. This item toggles the visibility of the tool bar.
- Status Bar Activated with mouse. This item toggles the visibility of the status bar.
- Birds Eye View Activated with mouse. This item toggles the visibility of the birds-eye view. <NOT IMPLEMENTED>.

Shape

This menubar pulldown is activated by clicking on 'Shape' in the menubar or by using the short cut key Meta-S.

- Group Activated with mouse or short cut key Ctrl-G (or key 'g' if the menu is visible). This item causes the selected shapes in the editor to be combined into one shape. Henceforth moving this one shape will move all it's constituent shapes and it can also be collapsed into a single icon. This item is disabled and dimmed if there is nothing selected in the editor.

- Ungroup Activated with mouse or short cut key Ctrl-U (or key 'u' if the menu is visible). This item causes the selected shapes in the editor to be decomposed into their constituent shapes. Henceforth these constituent shapes can be moved individually. This item is disabled and dimmed if there is nothing selected in the editor.
- Iconify Activated with mouse or short cut key Ctrl-T (or key 'c' if the menu is visible). This item causes the selected shapes in the editor to be grouped and then replaced with a single icon. This item is disabled and dimmed if there is nothing selected in the editor.
- DeIconify Activated with mouse or short cut key Ctrl-E (or key 'x' if the menu is visible). This item causes the selected shapes in the editor that were previously collapsed to be ungrouped. This item is disabled and dimmed if there is nothing selected in the editor.
- Bring to Front Activated with mouse or short cut key Ctrl-F (or key 'f' if the menu is visible). This item causes the selected shapes in the editor to be brought to the front of any deselected shapes. This item is disabled and dimmed if there is nothing selected in the editor.
- Send to Back Activated with mouse or short cut key Ctrl-B (or key 'b' if the menu is visible). This item causes the selected shapes in the editor to be sent to be behind any deselected shapes. This item is disabled and dimmed if there is nothing selected in the editor.
- Bring Forward Activated with mouse. This item causes each selected shape in the editor to be brought in front of the shape that is immediately in front of it. This item is disabled and dimmed if there is nothing selected in the editor.
- Send Backward Activated with mouse. This item causes each selected shape in the editor to be sent to be behind the shape immediately behind it. This item is disabled and dimmed if there is nothing selected in the editor.

Connect

This menubar pulldown is activated by clicking on 'Connect' in the menubar or by using the short cut key Meta-C.

- Connect Activated with mouse. This item causes each selected shape in the editor to be connected to every other selected shape in the editor. The type of connection is as specified in the toolbar. This item is disabled and dimmed if there is less than two nodes selected in the editor.
- Disconnect Activated with mouse. This item causes each selected shape in the editor to be disconnected from every other shape in the editor. This item is disabled and dimmed if there is nothing selected in the editor.

Format

This menubar pulldown is activated by clicking on 'Format' in the menubar or by using the short cut key Meta-O.

- Expand Editing Area Activated with mouse. This item causes the area in which shapes appear in the editor to be enlargened.
- Shrink Editing Area Activated with mouse. This item causes the area in which shapes appear in the editor to be reduced in size. This item only shrinks the area if it was previously expanded.
- AutoPlace Activated with mouse. This item causes an attempt to be made to place all nodes equidistant from each other and with a minimum of overlapping connections.

Help

This menubar pulldown is activated by clicking on 'Help' in the menubar or by using the short cut key Meta-H.

- About... Activated by selecting menu item. This should display a dialog box containing information about your application <NOT IMPLEMENTED>.
- Help Topics... Activated by selecting menu item. <NOT IMPLEMENTED>
- Tool Hints Activated by selecting menu item. This enables/disables tool hint help messages, which are displayed when the mouse cursor pauses over various widgets in the window. Note that tool hints are disabled in toolbars if the toolbar icons are already labeled.
- Balloon Help Activated by selecting menu item. This enables/disables balloon help messages, which are similar to tool hints but with more text and take longer to appear (at which time they replace any displayed tool hint). <NOT IMPLEMENTED>

Toolbars

Mica-supplied toolbars are similar to Mica menubars. There is a base functionality and then there is an implementation using that functionality that provides the most common features that can also be programmatically added to or removed from. The base functionality is provided by the MiToolBar class. The supplied implementation is provided by the MiEditorToolBar class.

Toolbar buttons can have a number of appearances consisting of your choice of border look coupled with the option to have the border appear for an tool only while it has mouse focus. Buttons can also have labels, if desired. The background menu automatically assigned to toolbars allows the user to interactively change this at runtime.

Toolbars can be orientated either vertically or horizontally and are dockable at any edge of a MiEditorWindow (and elsewhere, see MiDockingPanel).

Status Bars

Mica supplies the MiEditorStatusBar class, a subclass of MiStatusBar, that has been designed especially for drawing editors. It can contain a number of status bar *fields* and one *overlay* field. Status bar fields are areas within the status bar that display, sometimes editable, specialized data. The overlay field is usually invisible but, when visible, covers the entire length of the status bar. Any MiPart can be a status field or overlay field. However, a number of status fields are supplied with Mica:

- MiBasicStatusField
- MiCurrentTimeStatusField
- MiMagnificationStatusField
- MiMouseXYPositionStatusField
- MiStatusBarFocusStatusField
- MiSystemResourcesStatusField
- MiWhatsSelectedStatusField

Choosers

Mica supplies a number of choosers, special widgets that allow the end-user to select one of a number of possible values for an attribute. Some choosers are dialog boxes and some are option menus. Option menu choosers are commonly found in toolbars. The dialog choosers are:

- MiColorChooser
- MiFontChooser

The option menu choosers are:

- MiBorderLookOptionMenu
- MiColorOptionMenu
- MiFontOptionMenu
- MiFontPointSizeOptionMenu
- MiLineEndsOptionMenu

- MiLineWidthOptionMenu

Shape Attribute Dialog

This is a dialog window that displays and allows editing of the attributes of a MiPart, typically a ‘shape’. This dialog is similar to attribute dialogs found in most drawing programs.

Property Sheets

Mica supplies a number of classes for the development of *property sheets*. Property sheets are generally a user interface that allows the user to set the value of a number of named attributes. The most common property sheet is seen as two columns of text, the left column a list of names of attributes, the right column a list of (possibly editable) values. The class that implements this common property sheet is the MiBasicPropertyPanel class.

The MiComboPlusPropertyPanel is the same as the MiBasicPropertyPanel except that at the top of the sheet is a combo box, which would typically be used to allow the user to choose which object the property sheet is displaying properties of.

The MiListPlusPropertyPanel is the same as the MiBasicPropertyPanel except that at the left of the sheet is a scrolled list, which would typically be used to allow the user to choose which object the property sheet is displaying properties of.

The MiTablePropertyPanel is the same as the MiBasicPropertyPanel except that a MiTable is used to implement the sheet and the table allocates a column to each property and a row to each inspected object.

Property sheets are created by assigning a list of MiPropertyWidgets to MiPropertySheet. These are then used when the property sheet is *opened* as the widgets for the property panel. The MiPropertyWidget class supports initial sensitivity, dialog help and status bar help message generation. The MiPropertyWidget class saves a copy of the value of it’s property to support undo() (revert) and hasChanged() methods. It also saves a copy of the attributes of the widget so that the widget can temporarily change it’s attributes to indicate a validation error to the user.

The Class Hierarchy

MiWidget
    MiPropertyPanel

MiBasicPropertyPanel
MiComboPlusPropertyPanel
MiListPlusPropertyPanel
MiTablePropertyPanel

ClipBoard

The clipboard supports the cut, copy and pasting of MiParts. Paste operation The clipboard using the same API as drag-and-drop. Since this API is built-in to each MiPart support for the clipboard is automatically supported. In the future the clipboard will be a transparent gateway to the resident window system's clipboard.

Editor Background Menu

The editor background menu is a popup usually activated using the right mouse button that contains options to cut, or copy the selected items to the clipboard, to delete the selected items, to display the properties of the first selected item and to paste from the clipboard.

This background menu is automatically supplied to the current MiEditor in any MiEditorWindow.

The Class Hierarchy

MiiContextMenu
    MiEditorMenu
        MiEditorBackgroundMenu
MiCommand
    MiCommandWidgetCommand
        MiEditorBackgroundMenuCommands

End User Attributes Menu

This menu has options that allow an end-user to modify some of the attributes of an application while it is running. Each MiPart has an attribute that specifies which attributes are and are not available for modification by an end-user.

CHAPTER 15

Customizing: Properties, Styles and Prototypes

This chapter describes the support provided for customizing the look and feel of Mica.

About Customization

There are a range of approaches to customization in Mica, from the end-user typing in values of properties to customize text strings and attributes of an application to the programmer who has written a new toolkit to replace the default look-and-feels of all widgets.

Properties

Mica manages a tree of system-wide properties. The properties farther up in the tree are of higher priority than those nearer the root. The root properties have to do with the default properties used by Mica. Examples are any text (menu items, error messages, etc...), and icons (toolbars, message dialogs, etc...). The next level up contains properties specified as the default properties by the currently running application.

The next levels are loaded from text files. First looking in the user's home directory then in the local directory, each file found will have their properties loaded. The files examined are:

defaults.mica

This file typically contains text replacing the built in text strings, as desired, in perhaps another language.

properties.mica

This file typically contains properties that the end-user wants to use to customize their environment. For example:

```
Mi_IMAGES_HOME  = /home/my_better_images
```

would cause Mica to get all of the built-in icons it uses from the '/home/my_better_images' directory.

In addition the application can optionally load a property file dedicated to the application (for example MiLife.mica).

Printing all property names and values

All property names and values can be printed by pressing Ctrl-Shift-p, if enabled, or by calling:

```
new MiPrintGraphicsStructures().processCommand(MiPrintGraphicsStructures.PROPERTIES);
```

Macros

Any property name can be a macro (i.e. used as part of another properties value). The properties frequently used as macros are:

- Mi_HOME
- Mi_IMAGES_HOME
- Mi_CURRENT_DIRECTORY
- Mi_HOST_SYSTEM_ROOT_DIRECTORY
- Mi_VERSION

Macros can be nested to any depth. Macros are referenced as indicated by the following example:

```
Mi_IMAGES_HOME  = ${Mi_HOME}/images/
```

Internationalization (text, colors, images)

As noted above, all text strings are, and should be, values of some property. When text is assigned to a widget that displays text, the *name of the property* should be used, not the actual text string. For example

```
menu.setName(Mi_FILE_MENU_DISPLAY_NAME)
```

instead of

```
menu.setName("&File");
```

ALL text that is displayed by Mica is first checked to see if it is a property name, and if so, the value of the property is what is actually displayed. This is also true of ALL named colors and all named images.

Styles

Styles are implementations of the MiiCustomStyle interface. A style can be assigned to a individual MiWidget, to a class of MiWidgets (e.g. the MiPushButton class), or to the MiWidgets as a whole. At the end of each MiWidget constructor, if the MiWidget has a style, a call is made to the style's applyCustomStyle() method. In this method the widget can be modified as needed.

The sample implementation (MiCustomStyle) supports the adding of event handlers and action handlers to all widgets it is assigned to. For example this could be used to add an event handler (MiPlayEventSound) to all push buttons so that they 'buzz' when getting mouse focus.

Prototypes

MiWidgets are copied from a prototype when they are created. If it is desired to change the look and/or feel of a class of widgets, it is often easiest to modify the prototype. The copy process copies all event handlers and action handlers that have been assigned to the prototype.

Widget Factory

Each class of MiWidget have a static create() method (e.g. MiPushbutton.create()). This method creates a copy of the widget's prototype and applies any assigned styles to the widget, and then returns it. This provides a method by which the 'type' of widgets can be changed (to a subclass of the original widget). For example, one could do the following: MiPushButton.setPrototype(new MiSuperDuperPushButton()) and all successive calls to MiPushbutton.create() will return a new instance of MiSuperDuperPushButton.

CHAPTER 16

Debugging

This chapter describes debugging and how Mica supports debugging. Debugging techniques help you understand what parts of your application are working and what parts are not.

About Debugging

Mica supports debugging in a number of ways using assertions, traces that send output to a file and/or STDOUT, and internal sanity checks. The MiDebug class provides methods for you to customize many of the debugging capabilities of Mica.

- Assertions A number of assertions are made in key methods. These assertions verify things such as a MiPart's bounds being valid. If an assertion fail then a non-checked exception is thrown.
- Layout validity At a number of places in the code, the hierarchy of MiParts are checked for layout validity. If some part does not have a valid layout then a message is generated

identifying the problem and the MiPart involved. This problem usually is generated by a layout modifying the it's containers geometry. Layouts can be recalculated interactively by using the MiIReCalcLayouts event handler.

Tracing Many operations can be individually traced if enabled by using the MiDebug class (see *setTraceMode()*).

Event dispatch All event handler dispatching can be interactively traced by using the MiISetDebugTraceModes event handler.

Action dispatch All action generation can be traced by using the MiDebug.*traceActions*() method.

Structure dump The graphics container-part hierarchy can be printed by using the MiDebug.*dump*() methods or interactively using the of the MiIPrintGraphicsStructures event handler.

The MiDebug Class

The MiDebug class has a number of public static methods which support:

- Logging to a file
- Tracing
- Printing the trace stack
- Printing a MiPart and it's contents
- Printing dispatched actions

Special Debug Event Handlers

- MiISetDebugTraceModes

Ctrl-Shift-E Turn on/off tracing

Ctrl-Shift-D, 1 Trace translations of events into commands

Ctrl-Shift-D, 2 Trace keyboard focus and enter key focus assignments

Ctrl-Shift-D, 3 Trace drag and drop activity

Ctrl-Shift-D, 4 Trace event dispatching, short-cut (accelerators) event dispatching, event to command translation, event handler grabbing, and raw event input

Ctrl-Shift-D, 5 Trace interactive (end-user) selection and deselection of MiParts

- MiIPrintGraphicsStructures

Ctrl-Shift-F Dump to STDOUT and to the dbg.mica logging file the contents of the MiPart underneath the mouse cursor.

Ctrl-Shift-G Dump to STDOUT and to the dbg.mica logging file the contents of the window underneath the mouse cursor.

Ctrl-Shift-P Dump to STDOUT and to the dbg.mica logging file the contents of the entire properties table.

- MiIReCalcLayouts

Ctrl-Shift-L Invalidate the layouts of all MiParts in the window under the mouse cursor, causing them all to be re-validated and redrawn.

MiExceptionOccurredDialog

When an exception occurs in the event handling thread or the drawing thread a dialog window is displayed with three options: *Exit*, *Details* and, if the event handling thread, *Continue*. If Exit is selected the current program is exited. If Details is selected the stack trace is printed in the dialog. If Continue is chosen the thread is allowed to continue. Details are in all cases written to a file with a name of the form:

```
Error_trace_file_Thu_Apr_02_17_23_43_MST_1998
```

MiHierarchicalInspector

This is a dialog window that allows browsing of the structure and contents of a root window in Mica.

CHAPTER 17

Basic Types and Classes

This chapter describes some basic types and classes that are used throughout Mica.

About Coordinate Types

Mica uses a number of coordinate types. These types are and their corresponding Java types are:

MiCoord	->	double
MiDistance	->	double
MiDeviceCoord	->	int
MiDeviceDistance	->	int

For speed of compilation and execution and because Java does not support typedefs a preprocessing pass is used to convert these coordinate types into types that Java supports. This preprocessing pass is run by the makefile and uses a ‘sed’ script on Unix and a Java executable on other platforms.

About MiBounds

Instances of the MiBounds class are used wherever there is a need to specify a rectangular area in world space. The MiBounds class has a wealth of convenience methods to specify, inquire and modify MiBounds instances.

About MiSize

Instances of the MiSize class are used wherever there is a need to specify a rectangular dimension in world space. The MiSize class has a wealth of convenience methods to specify, inquire and modify MiSize instances.

About MiPoint

Instances of the MiPoint class are used wherever there is a need to specify a point in world space. The MiPoint class has a wealth of convenience methods to specify, inquire and modify MiPoint instances.

About MiVector

Instances of the MiVector class are used wherever there is a need to specify a 2 -dimensional distance in world space. The MiVector class has a wealth of convenience methods to specify, inquire and modify MiVector instances.

About MiScale

Instances of the MiScale class are used wherever there is a need to specify a point in world space. The MiScale class has a wealth of convenience methods to specify, inquire and modify MiScale instances.

About MiDeviceBounds

Instances of the MiDeviceBounds class are used wherever there is a need to specify a rectangular area in device space. The MiDeviceBounds class has a wealth of convenience methods to specify, inquire and modify MiDeviceBounds instances.

About MiDevicePoint

Instances of the MiDevicePoint class are used wherever there is a need to specify a point in device space. The MiDevicePoint class has a wealth of convenience methods to specify, inquire and modify MiDevicePoint instances.

About MiDeviceVector

Instances of the MiDeviceVector class are used wherever there is a need to specify a 2 -dimensional distance in device space. The MiDeviceVector class has a wealth of convenience methods to specify, inquire and modify MiDeviceVector instances.

About Attachments

MiParts can have attachments, which are other MiParts. This is in essence the parts private container-part hierarchy. This is often used when one wants to temporarily associate a part with another MiPart (for example the ‘handles’ associated with a selected shape in a drawing editor). There are a number of methods of the MiPart class which support the adding, removing and inquiring of attachments.

CHAPTER 18

Special Topics

This chapter describes details about topics that are necessary for a deep understanding of Mica.

Drag and drop

To do.

Picking

To do.

Caveats

To do.

CHAPTER 19

Advanced Topic: Transforms

This chapter describes transforms and how and why they are used. You can use transforms to programmatically modify the scales and translations of MiParts.

About Transforms

To do.

CHAPTER 20

Advanced Topic: Renderers

This chapter describes renderers and how and why they are used. You can use renderers to customize how MiParts are drawn,

About Renderers

To do.

CHAPTER 21

Attribute Tables

This chapter contains tables that describe the attributes associated with each MiPart.

The Tables

TABLE 1. Background Image

Description	Specifies what Image, if any, to draw in the interior of the MiPart. The image is resized, if necessary, to the bounds of the MiPart. The image is truncated, if necessary, to the boundaries of the MiPart <NOT IMPLEMENTED JDK 1.0.2>.
MiPart Methods	setBackgroundImage(Image) Image getBackgroundImage()

TABLE 1. Background Image

Valid Values	Any valid java.awt.Image, null
Default Value	null
Caveats	The background image will not be displayed if the background color is equals to Mi_TRANSPARENT_COLOR. The background color is equal to Mi_TRANSPARENT_COLOR by default.
Name	Mi_BACKGROUND_IMAGE_ATT_NAME
Key	Mi_BACKGROUND_IMAGE
See Also	Background Tile, Fill Color

TABLE 2. Background Tile

Description	Specifies what Image, if any, to draw in the interior of the MiPart. The image is not resized but is replicated, if necessary, row by row and column by column, to fill the bounds of the MiPart. The image is truncated, if necessary, to the boundaries of the MiPart.
MiPart Methods	setBackgroundTile(Image) Image getBackgroundTile()
Valid Values	Any valid java.awt.Image, null
Default Value	null
Caveats	
Name	Mi_BACKGROUND_TILE_ATT_NAME
Key	Mi_BACKGROUND_TILE
See Also	Background Image

TABLE 3. **Font**

Description	Specifies what Font to use when any text associated with the MiPart is drawn.
MiPart Methods	setFont(MiFont) MiFont getFont() setFontBold(boolean) boolean isFontBold() setFontItalic(boolean) boolean isFontItalic() setFontPointSize(int) int getFontPointSize()
Valid Values	Any valid MiFont
Default Value	MiAttributes.defaultFont
Caveats	
Name	Mi_FONT_ATT_NAME
Key	Mi_FONT
See Also	

TABLE 4. **Tool Hint Help**

Description	Specifies the content and appearance of the tool hint, if any, to be associated with the MiPart. A tool hint is the small, usually single line text message that appear when the user pauses the mouse cursor over the MiPart. The content can be any text string and the appearance can be any MiAttributes.
MiPart Methods	setToolHintHelp(MiiHelpInfo info) MiiHelpInfo getToolHintHelp() setToolHintMessage(String msg)
Valid Values	Any valid MiiHelpInfo, String, null
Default Value	null
Caveats	

TABLE 4. Tool Hint Help

Name	Mi_TOOL_HINT_HELP_ATT_NAME
Key	Mi_TOOL_HINT_HELP
See Also	
See Also (Classes)	MiIDisplayToolHints

TABLE 5. Balloon Help

Description	Specifies the content and appearance of the balloon help, if any, to be associated with the MiPart. Balloon help is the stylized callout, usually large multi-line message that appear when the user pauses the mouse cursor over the MiPart. The content can be any text string and the appearance can be any MiAttributes.
MiPart Methods	setBalloonHelp(MiiHelpInfo info) MiiHelpInfo getBalloonHelp() setBalloonMessage(String msg)
Valid Values	Any valid MiiHelpInfo, String, null
Default Value	null
Caveats	
Name	Mi_BALLOON_HELP_ATT_NAME
Key	Mi_BALLOON_HELP
See Also	Tool Hint Help

TABLE 6. Status Help

Description	Specifies the content and appearance of the status help, if any, to be associated with the MiPart. Status help is the message displayed in the status bar when the user moves the mouse cursor over the MiPart. The content can be any text string and the appearance can be any MiAttributes.
MiPart Methods	setStatusHelp(MiiHelpInfo info) MiiHelpInfo getStatusHelp() setStatusHelpMessage(String msg)
Valid Values	Any valid MiiHelpInfo, String, null
Default Value	null
Caveats	
Name	Mi_STATUS_HELP_ATT_NAME
Key	Mi_STATUS_HELP
See Also	
See Also (Classes)	MiStatusBarFocusManager

TABLE 7. Dialog Help

Description	Specifies the content and appearance of the dialog help, if any, to be associated with the MiPart. Dialog help is a dialog box displayed in response to the user pressing the help key over the MiPart. The content can be any text string and the appearance can be any MiAttributes.
MiPart Methods	setDialogHelp(MiiHelpInfo info) MiiHelpInfo getDialogHelp() setDialogMessage(String msg)
Valid Values	Any valid MiiHelpInfo, String, null
Default Value	null
Caveats	
Name	Mi_DIALOG_HELP_ATT_NAME
Key	Mi_DIALOG_HELP

TABLE 7. Dialog Help

See Also	
See Also (Classes)	MiIDisplayHelpDialog

TABLE 8. Shadow Renderer

Description	Specifies the renderer that will draw the shadow(s) for the MiPart.
MiPart Methods	setShadowRenderer(MiiShadowRenderer) MiiShadowRenderer getShadowRenderer()
Valid Values	Any valid MiiShadowRenderer, null
Default Value	null
Caveats	
Name	Mi_SHADOW_RENDERER_ATT_NAME
Key	Mi_SHADOW_RENDERER
See Also	
See Also (Classes)	MiShadowRenderer

TABLE 9. Before Renderer

Description	Specifies the renderer that will be called to draw *before* the MiPart is drawn.
MiPart Methods	setBeforeRenderer(MiiPartRenderer) MiiPartRenderer getBeforeRenderer()
Valid Values	Any valid MiiPartRenderer, null
Default Value	null
Caveats	
Name	Mi_BEFORE_RENDERER_ATT_NAME
Key	Mi_BEFORE_RENDERER
See Also	
See Also (Classes)	MiPartRenderer

TABLE 10. After Renderer

Description	Specifies the renderer that will be called to draw *after* the MiPart is drawn.
MiPart Methods	setAfterRenderer(MiiPartRenderer) MiiPartRenderer getAfterRenderer()
Valid Values	Any valid MiiPartRenderer, null
Default Value	null
Caveats	
Name	Mi_AFTER_RENDERER_ATT_NAME
Key	Mi_AFTER_RENDERER
See Also	
See Also (Classes)	MiPartRenderer

TABLE 11. Line Ends Renderer

Description	Specifies the renderer that will be called to draw any line endpoints (for example arrow heads) that may be associated with the MiPart.
MiPart Methods	setLineEndsRenderer(MiiLineEndsRenderer) MiiLineEndsRenderer getLineEndsRenderer()
Valid Values	Any valid MiiLineEndsRenderer, null
Default Value	MiAttributes.defaultLineEndsRenderer
Caveats	
Name	Mi_LINE_ENDS_RENDERER_ATT_NAME
Key	Mi_LINE_ENDS_RENDERER
See Also	Line Start Style, Line Start Size, Line End Style, Line End Size
See Also (Classes)	MiLineEndsRenderer

TABLE 12. Connection Point Manager

Description	Specifies the manager that is responsible for determining the locations of custom connection points of the MiPart.
MiPart Methods	setConnectionPointManager(MiConnectionPointManager) MiConnectionPointManager getConnectionPointManager()
Valid Values	Any valid MiConnectionPointManager, null
Default Value	null
Caveats	
Name	Mi_CONNECTION_POINT_MANAGER_ATT_NAME
Key	Mi_CONNECTION_POINT_MANAGER
See Also	
See Also (Classes)	MiConnectionPointManager

TABLE 13. Background Renderer

Description	Specifies the renderer that will be called to draw the background of the MiPart. This is used when something more complex than a solid color or Image fill is desired as the background of the MiPart
MiPart Methods	setBackgroundRenderer(MiiDeviceRenderer) MiiDeviceRenderer getBackgroundRenderer()
Valid Values	Any valid MiiDeviceRenderer, null
Default Value	null
Caveats	
Name	Mi_BACKGROUND_RENDERER_ATT_NAME
Key	Mi_BACKGROUND_RENDERER
See Also	
See Also (Classes)	MiiDeviceRenderer

TABLE 14. Border Renderer

Description	Specifies the renderer that will be called to draw any border look and/or border hilite assigned to the MiPart.
MiPart Methods	setBorderRenderer(MiiDeviceRenderer) MiiDeviceRenderer getBorderRenderer()
Valid Values	Any valid MiiDeviceRenderer, null
Default Value	Miattributes.defaultBorderRenderer
Caveats	
Name	Mi_BORDER_RENDERER_ATT_NAME
Key	Mi_BORDER_RENDERER
See Also	Border Look, Border Hilite Color, Border Hilite Width, Has Border Hilite
See Also (Classes)	MiiDeviceRenderer, MiBorderLookRenderer

TABLE 15. Visibility Animator

Description	Specifies the animator that will be called to animate the appearance and disappearance of the MiPart.
MiPart Methods	setVisibilityAnimator(MiPartAnimator) MiPartAnimator setVisibilityAnimator()
Valid Values	Any valid MiPartAnimator, null
Default Value	null
Caveats	
Name	Mi_VISIBILITY_ANIMATOR_ATT_NAME
Key	Mi_VISIBILITY_ANIMATOR
See Also	
See Also (Classes)	MiPartAnimator

TABLE 16. Context Menu

Description	Specifies the menu that will be displayed in response to the user pressing the menu popup key (usually the right mouse button) over the MiPart.
MiPart Methods	setContextMenu(MiiContextMenu) MiiContextMenu getContextMenu()
Valid Values	Any valid MiiContextMenu, null
Default Value	null
Caveats	
Name	Mi_CONTEXT_MENU_ATT_NAME
Key	Mi_CONTEXT_MENU
See Also	Conext Cursor
See Also (Classes)	MiiContextMenu, MiIDisplayContextMenu, MiEditorMenu

TABLE 17. Color

Description	Specifies the color that will be used to draw the MiPart. Also called *foreground* color by some people.
MiPart Methods	setColor(Color) setColor(String) Color getColor()
Valid Values	Any valid Color, MiiTypes.Mi_TRANSPARENT_COLOR, any valid color name (see MiColorManager), RGB specified by prefix “0x” or “#”
Default Value	MiColorManager.black
Caveats	
Name	Mi_COLOR_ATT_NAME
Key	Mi_COLOR
See Also	Background Color
See Also (Classes)	MiColorManager, java.awt.Color

TABLE 18. Background Color

Description	Specifies the background color that will be used to draw the MiPart. Also called *fill* color by some people.
MiPart Methods	setBackgroundColor(Color) setBackgroundColor(String) Color getBackgroundColor()
Valid Values	Any valid Color, MiiTypes.Mi_TRANSPARENT_COLOR, any valid color name (see MiColorManager), RGB specified by prefix “0x” or “#”
Default Value	MiiTypes.Mi_TRANSPARENT_COLOR
Caveats	
Name	Mi_BACKGROUND_COLOR_ATT_NAME
Key	Mi_BACKGROUND_COLOR
See Also	Color
See Also (Classes)	MiColorManager, java.awt.Color

TABLE 19. White Color

Description	Specifies the white (brightest) color that will be used to draw the MiPart. This is typically used when drawing a beveled border for the MiPart.
MiPart Methods	setWhiteColor(Color) Color getWhiteColor()
Valid Values	Any valid Color, MiiTypes.Mi_TRANSPARENT_COLOR
Default Value	MiColorManager.white
Caveats	
Name	Mi_WHITE_COLOR_ATT_NAME
Key	Mi_WHITE_COLOR
See Also	Light Color
See Also (Classes)	MiColorManager

TABLE 20. Light Color

Description	Specifies the light (second brightest) color that will be used to draw the MiPart. This is typically used when drawing a beveled border for the MiPart.
MiPart Methods	setLightColor(Color) Color getLightColor()
Valid Values	Any valid Color, MiiTypes.Mi_TRANSPARENT_COLOR
Default Value	MiColorManager.lightGray2
Caveats	
Name	Mi_LIGHT_COLOR_ATT_NAME
Key	Mi_LIGHT_COLOR
See Also	White Color
See Also (Classes)	MiColorManager

TABLE 21. Dark Color

Description	Specifies the dark (third brightest) color that will be used to draw the MiPart. This is typically used when drawing a beveled border for the MiPart.
MiPart Methods	setDarkColor(Color) Color getDarkColor()
Valid Values	Any valid Color, MiiTypes.Mi_TRANSPARENT_COLOR
Default Value	MiColorManager.gray
Caveats	
Name	Mi_DARK_COLOR_ATT_NAME
Key	Mi_DARK_COLOR
See Also	Black Color
See Also (Classes)	MiColorManager

TABLE 22. Black Color

Description	Specifies the darkest color that will be used to draw the MiPart. This is typically used when drawing a beveled border for the MiPart.
MiPart Methods	setBlackColor(Color) Color getBlackColor()
Valid Values	Any valid Color, MiiTypes.Mi_TRANSPARENT_COLOR
Default Value	MiColorManager.darkGray
Caveats	
Name	Mi_BLACK_COLOR_ATT_NAME
Key	Mi_BLACK_COLOR
See Also	Dark Color
See Also (Classes)	MiColorManager

TABLE 23. Border Hilite Color

Description	Specifies the color that will be used to draw the hilite border, if any, of the MiPart.
MiPart Methods	setBorderHiliteColor(Color) Color getBorderHiliteColor()
Valid Values	Any valid Color, MiiTypes.Mi_TRANSPARENT_COLOR
Default Value	MiColorManager.black
Caveats	
Name	Mi_BORDER_HILITE_COLOR_ATT_NAME
Key	Mi_BORDER_HILITE_COLOR
See Also	Has Border Hilite
See Also (Classes)	MiColorManager

TABLE 24. Border Look

Description	Specifies the look of the border drawn around subclasses of MiVisibleContainer (i.e. all MiWidgets) and the look of the lines drawn in shapes (i.e. lines, ovals and rectangle, etc.).
MiPart Methods	setBorderLook(int) int getBorderLook()
Valid Values	Mi_FLAT_BORDER_LOOK Mi_NO_BORDER_LOOK Mi_RAISED_BORDER_LOOK Mi_INDENTED_BORDER_LOOK Mi_GROOVE_BORDER_LOOK Mi_RIDGE_BORDER_LOOK Mi_OUTLINED_RAISED_BORDER_LOOK Mi_OUTLINED_INDENTED_BORDER_LOOK Mi_INLINED_RAISED_BORDER_LOOK Mi_INLINED_INDENTED_BORDER_LOOK Mi_SQUARE_RAISED_BORDER_LOOK
Default Value	Mi_FLAT_BORDER_LOOK
Caveats	
Name	Mi_BORDER_LOOK_ATT_NAME
Key	Mi_BORDER_LOOK
See Also	White, Light, Dark and Black Color

TABLE 25. Line Style

Description	Specifies how any lines associated with the MiPart will be drawn.
MiPart Methods	setLineStyle(int) int getLineStyle()

TABLE 25. Line Style

Valid Values	Mi_SOLID_LINE_STYLE Mi_DASHED_LINE_STYLE <NOT IMPLEMENTED> Mi_DOUBLE_DASHED_LINE_STYLE <NOT IMPLEMENTED>
Default Value	Mi_SOLID_LINE_STYLE
Caveats	
Name	Mi_LINE_STYLE_ATT_NAME
Key	Mi_LINE_STYLE
See Also	

TABLE 26. Line Start Style

Description	Specifies how the start of any line associated with the MiPart will be drawn (for example an *arrow tail*)
MiPart Methods	setLineStartStyle(int) int getLineStartStyle()

TABLE 26. Line Start Style

Valid Values	Mi_NONE Mi_FILLED_TRIANGLE_LINE_END_STYLE Mi_THIN_ARROW_LINE_END_STYLE Mi_THICK_ARROW_LINE_END_STYLE Mi_FILLED_CIRCLE_LINE_END_STYLE Mi_FILLED_SQUARE_LINE_END_STYLE Mi_TRIANGLE_VIA_LINE_END_STYLE Mi_FILLED_TRIANGLE_VIA_LINE_END_STYLE Mi_CIRCLE_VIA_LINE_END_STYLE Mi_FILLED_CIRCLE_VIA_LINE_END_STYLE Mi_SQUARE_VIA_LINE_END_STYLE Mi_FILLED_SQUARE_VIA_LINE_END_STYLE Mi_TRIANGLE_LINE_END_STYLE Mi_CIRCLE_LINE_END_STYLE Mi_SQUARE_LINE_END_STYLE Mi_DIAMOND_LINE_END_STYLE Mi_FILLED_DIAMOND_LINE_END_STYLE Mi_3FEATHER_LINE_END_STYLE Mi_2FEATHER_LINE_END_STYLE
Default Value	Mi_NONE
Caveats	
Name	Mi_LINE_START_STYLE_ATT_NAME
Key	Mi_LINE_START_STYLE
See Also	Line Start Size, Line End Style, Line End Size, Line Ends Renderer
See Also (Classes)	MiLineEndsRenderer

TABLE 27. Line End Style

Description	Specifies how the end of any line associated with the MiPart will be drawn (for example an *arrow head*)
MiPart Methods	setLineEndStyle(int) int getLineEndStyle()
Valid Values	Mi_NONE Mi_FILLED_TRIANGLE_LINE_END_STYLE Mi_THIN_ARROW_LINE_END_STYLE Mi_THICK_ARROW_LINE_END_STYLE Mi_FILLED_CIRCLE_LINE_END_STYLE Mi_FILLED_SQUARE_LINE_END_STYLE Mi_TRIANGLE_VIA_LINE_END_STYLE Mi_FILLED_TRIANGLE_VIA_LINE_END_STYLE Mi_CIRCLE_VIA_LINE_END_STYLE Mi_FILLED_CIRCLE_VIA_LINE_END_STYLE Mi_SQUARE_VIA_LINE_END_STYLE Mi_FILLED_SQUARE_VIA_LINE_END_STYLE Mi_TRIANGLE_LINE_END_STYLE Mi_CIRCLE_LINE_END_STYLE Mi_SQUARE_LINE_END_STYLE Mi_DIAMOND_LINE_END_STYLE Mi_FILLED_DIAMOND_LINE_END_STYLE Mi_3FEATHER_LINE_END_STYLE Mi_2FEATHER_LINE_END_STYLE
Default Value	Mi_NONE
Caveats	
Name	Mi_LINE_END_STYLE_ATT_NAME
Key	Mi_LINE_END_STYLE

TABLE 27. Line End Style

See Also	Line End Size, Line Start Style, Line Start Size, Line Ends Renderer
See Also (Classes)	MiLineEndsRenderer

TABLE 28. Write Mode

Description	Specifies how colors of the MiPart will be mixed with the colors already on the output device when the MiPart is drawn.
MiPart Methods	setWriteMode(int) int getWriteMode()
Valid Values	Mi_COPY_WRITEMODE (replace the colors of the pixels in the output buffer with the colors of the pixels of the MiPart) Mi_XOR_WRITEMODE (xor the colors of the pixels in the output buffer with the colors of the pixels of the MiPart)
Default Value	Mi_COPY_WRITEMODE
Caveats	
Name	Mi_WRITE_MODE_ATT_NAME
Key	Mi_WRITE_MODE
See Also	

TABLE 29. Context Cursor

Description	Specifies how the shape of the mouse cursor will appear while it is over the MiPart.
MiPart Methods	setContextCursor(int) int getContextCusror()

TABLE 29. Context Cursor

Valid Values	MiNONE <NOT IMPLEMENTED> Mi_DEFAULT_CURSOR Mi_CROSSHAIR_CURSOR Mi_TEXT_CURSOR Mi_WAIT_CURSOR Mi_SW_RESIZE_CURSOR Mi_SE_RESIZE_CURSOR Mi_NW_RESIZE_CURSOR Mi_NE_RESIZE_CURSOR Mi_N_RESIZE_CURSOR Mi_S_RESIZE_CURSOR Mi_W_RESIZE_CURSOR Mi_E_RESIZE_CURSOR Mi_HAND_CURSOR Mi_MOVE_CURSOR
Default Value	Mi_DEFAULT_CURSOR
Caveats	
Name	Mi_CONTEXT_CURSOR_ATT_NAME
Key	Mi_CONTEXT_CURSOR
See Also	
See Also (Classes)	java.awt.Frame

TABLE 30. **Minimum Width**

Description	Specifies the minimum width of the MiPart.
MiPart Methods	setMinimumWidth(MiDistance) MiDistance getMinimumWidth()
Valid Values	>= 0
Default Value	0
Caveats	The minimum width is not enforeced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_MINIMUM_WIDTH_ATT_NAME
Key	Mi_MINIMUM_WIDTH
See Also	Minimum Height, Maximum Width

TABLE 31. **Minimum Height**

Description	Specifies the minimum height of the MiPart.
MiPart Methods	setMinimumHeight(MiDistance) MiDistance getMinimumHeight()
Valid Values	>= 0
Default Value	0
Caveats	The minimum height is not enforeced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_MINIMUM_HEIGHT_ATT_NAME
Key	Mi_MINIMUM_HEIGHT
See Also	Minimum Width, Maximum Height

TABLE 32. Maximum Width

Description	Specifies the maximum width of the MiPart.
MiPart Methods	setMaximumWidth(MiDistance) MiDistance getMaximumWidth()
Valid Values	>= 0
Default Value	MAX_DISTANCE_VALUE
Caveats	The maximum width is not enforeced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_MAXIMUM_WIDTH_ATT_NAME
Key	Mi_MAXIMUM_WIDTH
See Also	Maximum Height, Minimum Width

TABLE 33. Maximum Height

Description	Specifies the maximum height of the MiPart.
MiPart Methods	setMaximumHeight(MiDistance) MiDistance getMaximumHeight()
Valid Values	>= 0
Default Value	MAX_DISTANCE_VALUE
Caveats	The maximum height is not enforeced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_MAXIMUM_HEIGHT_ATT_NAME
Key	Mi_MAXIMUM_HEIGHT
See Also	Minimum Height, Maximum Width

TABLE 34. Border Hilite Width

Description	Specifies the width of the hilite border, if any, of the MiPart.
MiPart Methods	setBorderHiliteWidth(MiDistance) MiDistance getBorderHiliteWidth()
Valid Values	>= 0
Default Value	2.0
Caveats	
Name	Mi_BORDER_HILITE_WIDTH_ATT_NAME
Key	Mi_BORDER_HILITE_WIDTH
See Also	Has Border Hilite, Border Hilite Color

TABLE 35. Line Width

Description	Specifies the width of any lines associated with the MiPart.
MiPart Methods	setLineWidth(MiDistance) MiDistance getLineWidth()
Valid Values	>= 0
Default Value	0
Caveats	
Name	Mi_LINE_WIDTH_ATT_NAME
Key	Mi_LINE_WIDTH
See Also	Line Style, Color

TABLE 36. Line Start Size

Description	Specifies the size of any line start style assigned to the MiPart.
MiPart Methods	setLineStartSize(MiDistance) MiDistance getLineStartSize()

TABLE 36. Line Start Size

Valid Values	>= 0
Default Value	10.0
Caveats	
Name	Mi_LINE_START_SIZE_ATT_NAME
Key	Mi_LINE_START_SIZE
See Also	Line Start Style, Line Ends Size Fn of Line Width
See Also (Classes)	MiLineEndsRenderer

TABLE 37. Line End Size

Description	Specifies the size of any line end style assigned to the MiPart.
MiPart Methods	setLineEndSize(MiDistance) MiDistance getLineEndSize()
Valid Values	>= 0
Default Value	10.0
Caveats	
Name	Mi_LINE_END_SIZE_ATT_NAME
Key	Mi_LINE_END_SIZE
See Also	Line End Style, Line Ends Size Fn of Line Width
See Also (Classes)	MiLineEndsRenderer

TABLE 38. Deletable

Description	Specifies whether the MiPart can be deleted.
MiPart Methods	setDeletable(boolean) boolean isDeletable()

TABLE 38. Deletable

Valid Values	true, false
Default Value	true
Caveats	This attribute is not enforeced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_DELETABLE_ATT_NAME
Key	Mi_DELETABLE
See Also	

TABLE 39. Movable

Description	Specifies whether the MiPart can be moved.
MiPart Methods	setMovable(boolean) boolean isMovable()
Valid Values	true, false
Default Value	true
Caveats	This attribute is not enforeced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_MOVABLE_ATT_NAME
Key	Mi_MOVABLE
See Also	

TABLE 40. Copyable

Description	Specifies whether the MiPart can be copied.
MiPart Methods	setCopyable(boolean) boolean isCopyable()
Valid Values	true, false
Default Value	true
Caveats	This attribute is not enforeced by the MiPart. It should be enforced by the manipulators of the MiPart.

TABLE 40. Copyable

Name	Mi_COPYABLE_ATT_NAME
Key	Mi_COPYABLE
See Also	

TABLE 41. Selectable

Description	Specifies whether the MiPart can be selected.
MiPart Methods	setSelectable(boolean) boolean isSelectable()
Valid Values	true, false
Default Value	true
Caveats	This attribute is not enforeced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_SELECTABLE_ATT_NAME
Key	Mi_SELECTABLE
See Also	

TABLE 42. Fixed Width

Description	Specifies whether the MiPart has a contant horizontal size.
MiPart Methods	setFixedWidth(boolean) boolean hasFixedWidth()
Valid Values	true, false
Default Value	false
Caveats	This attribute is not enforeced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_FIXED_WIDTH_ATT_NAME
Key	Mi_FIXED_WIDTH
See Also	Fixed Height, Minimum Width, Maximum Width

TABLE 43. Fixed Height

Description	Specifies whether the MiPart has a contant vertical size.
MiPart Methods	setFixedHeight(boolean) boolean hasFixedHeight()
Valid Values	true, false
Default Value	false
Caveats	This attribute is not enforeced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_FIXED_HEIGHT_ATT_NAME
Key	Mi_FIXED_HEIGHT
See Also	Fixed Width, Minimum Height, Maximum Height

TABLE 44. Fixed Aspect Ratio

Description	Specifies whether the MiPart has a constant horizontal size to vertical size ratio.
MiPart Methods	setFixedAspectRatio(boolean) boolean hasFixedAspectRatio()
Valid Values	true, false
Default Value	false
Caveats	This attribute is not enforeced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_FIXED_ASPECT_RATIO_ATT_NAME
Key	Mi_FIXED_ASPECT_RATIO
See Also	

TABLE 45. Attribute Lock Mask

Description	Specifies what attributes of the MiPart can *not* be changed.
MiPart Methods	setAttributeLockMask(int) int getAttributeLockMask()
Valid Values	Mi_NONE or any combination of: Mi_COLOR_ATTRIBUTE_MASK_BIT Mi_BACKGROUND_COLOR_ATTRIBUTE_MASK_BIT Mi_LINE_WIDTH_ATTRIBUTE_MASK_BIT Mi_WRITE_MODE_ATTRIBUTE_MASK_BIT Mi_FONT_ATTRIBUTE_MASK_BIT
Default Value	Mi_NONE
Caveats	This attribute is not enforced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_ATTRIBUTE_LOCK_MASK_ATT_NAME
Key	Mi_ATTRIBUTE_LOCK_MASK
See Also	

TABLE 46. Attribute Public Mask

Description	Specifies what attributes of the MiPart can be changed by the *end user.* This attribute is to be used by any end user menus that allow the user to change things like color and fonts, interactively (see MiEndUserAttsPopupMenu).
MiPart Methods	setAttributePublicMask(int) int getAttributePublicMask()

TABLE 46. Attribute Public Mask

Valid Values	Mi_NONE or any combination of: Mi_COLOR_ATTRIBUTE_MASK_BIT Mi_BACKGROUND_COLOR_ATTRIBUTE_MASK_BIT Mi_LINE_WIDTH_ATTRIBUTE_MASK_BIT Mi_WRITE_MODE_ATTRIBUTE_MASK_BIT Mi_FONT_ATTRIBUTE_MASK_BIT
Default Value	Mi_NONE
Caveats	This attribute is not enforeced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_ATTRIBUTE_LOCK_MASK_ATT_NAME
Key	Mi_ATTRIBUTE_LOCK_MASK
See Also	Attribute Lock Mask
See Also (Classes)	MiEndUserAttsPopupMenu

TABLE 47. Pickable

Description	Specifies whether the MiPart can be picked.
MiPart Methods	setPickable(boolean) boolean isPickable()
Valid Values	true, false
Default Value	true
Caveats	
Name	Mi_PICKABLE_ATT_NAME
Key	Mi_PICKABLE
See Also	

TABLE 48. Ungroupable

Description	Specifies whether the MiPart can be ungrouped (separated into constituent MiParts).
MiPart Methods	setUngroupable(boolean) boolean isUngroupable()
Valid Values	true, false
Default Value	true
Caveats	This attribute is not enforeced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_UNGROUPABLE_ATT_NAME
Key	Mi_UNGROUPABLE
See Also	

TABLE 49. Connectable

Description	Specifies whether the MiPart can be connected to.
MiPart Methods	setConnectable(boolean) boolean isConnectable()
Valid Values	true, false
Default Value	true
Caveats	This attribute is not enforeced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_CONNECTABLE_ATT_NAME
Key	Mi_CONNECTABLE
See Also	

TABLE 50. Hidden

Description	Specifies whether the MiPart is hidden (e.g. not visible but still having bounds and taking up space on the output device).
MiPart Methods	setHidden(boolean) boolean isHidden()
Valid Values	true, false
Default Value	false
Caveats	This attribute is not enforeced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_HIDDEN_ATT_NAME
Key	Mi_HIDDEN
See Also	

TABLE 51. Drag and Drop Source

Description	Specifies whether the MiPart is a data source for drag and drop operations.
MiPart Methods	setIsDragAndDropSource(boolean) boolean isDragAndDropSource()
Valid Values	true, false
Default Value	false
Caveats	
Name	Mi_DRAG_AND_DROP_SOURCE_ATT_NAME
Key	Mi_DRAG_AND_DROP_SOURCE
See Also	Drag and Drop Target
See Also (Classes)	MiDragAndDropManager, MiDragAndDropBehavior, MiiDragAndDropBehavior

TABLE 52. Drag and Drop Target

Description	Specifies whether the MiPart is a data target for drag and drop operations.
MiPart Methods	setIsDragAndDropTarget(boolean) boolean isDragAndDropTarget()
Valid Values	true, false
Default Value	false
Caveats	
Name	Mi_DRAG_AND_DROP_TARGET_ATT_NAME
Key	Mi_DRAG_AND_DROP_TARGET
See Also	Drag And Drop Source
See Also (Classes)	MiDragAndDropManager, MiDragAndDropBehavior, MiiDragAndDropBehavior

TABLE 53. Accepting Mouse Focus

Description	Specifies whether the MiPart accepts mouse focus.
MiPart Methods	setAcceptingMouseFocus(boolean) boolean isAcceptingMouseFocus()
Valid Values	true, false
Default Value	false
Caveats	This attribute is not enforced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_ACCEPTS_MOUSE_FOCUS_ATT_NAME
Key	Mi_ACCEPTS_MOUSE_FOCUS
See Also	

TABLE 54. Accepting Keyboard Focus

Description	Specifies whether the MiPart accepts keyboard focus.
MiPart Methods	setAcceptingKeyboardFocus(boolean) boolean isAcceptingKeyboardFocus()
Valid Values	true, false
Default Value	false
Caveats	This attribute is not enforced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_ACCEPTS_KEYBOARD_FOCUS_ATT_NAME
Key	Mi_ACCEPTS_KEYBOARD_FOCUS
See Also	

TABLE 55. Accepting Enter Key Focus

Description	Specifies whether the MiPart accepts enter key focus.
MiPart Methods	setAcceptingEnterKeyFocus(boolean) boolean isAcceptingEnterKeyFocus()
Valid Values	true, false
Default Value	false
Caveats	This attribute is not enforced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_ACCEPTS_ENTER_KEY_FOCUS_ATT_NAME
Key	Mi_ACCEPTS_ENTER_KEY_FOCUS
See Also	

TABLE 56. Accepting Tab Keys

Description	Specifies whether the MiPart accepts tab keys.
MiPart Methods	setAcceptingTabKeys(boolean) boolean isAcceptingTabKeys()
Valid Values	true, false
Default Value	true
Caveats	This attribute is not enforced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_ACCEPTS_TAB_KEYS_ATT_NAME
Key	Mi_ACCEPTS_TAB_KEYS
See Also	

TABLE 57. Has Border Hilite

Description	Specifies whether the MiPart has a hilite border.
MiPart Methods	setHasBorderHilite(boolean) boolean getHasBorderHilite()
Valid Values	true, false
Default Value	false
Caveats	This attribute is not enforced by the MiPart. It should be enforced by the manipulators of the MiPart.
Name	Mi_HAS_BORDER_HILITE_ATT_NAME
Key	Mi_HAS_BORDER_HILITE
See Also	

TABLE 58. Line Ends Size a Function of Line Width

Description	Specifies whether the any Line Starts or Ends of the MiPart automatically resize is response to changes in the line width attribute of the MiPart.
MiPart Methods	setLineEndsSizeFnOfLineWidth(boolean) boolean getLineEndsSizeFnOfLineWidth()
Valid Values	true, false
Default Value	true
Caveats	
Name	Mi_LINE_ENDS_SIZE_FN_OF_LINE_WIDTH_ATT_NAME
Key	Mi_LINE_ENDS_SIZE_FN_OF_LINE_WIDTH
See Also	MiLineEndsRenderer class

CHAPTER 22

Colors

This chapter describes the named colors supported by Mica.

About Colors

At this time Mica uses java.AWT.Color as the color in all APIs. However this may change to better support partial transparency.

Color Names

Mica supports a limited set of *named* colors. Other colors can also be specified using a text string by using the hexadecimal format to indicate the RGB value of the color (for example "0xff0000" or "#ff0000"). The 100% transparent (i.e. completely invisible) color can be specified with the name

"transparent" (MiiTypes.Mi_TRANSPARENT_COLOR_NAME) and has a color equal to null (MiiTypes.Mi_TRANSPARENT_COLOR).

TABLE 59. **Color Names**

Name	RGB (Red, green, Blue)	Browser Safe	Found In awt.Color
Black	0, 0, 0	Y	Y
darkGray	51, 51, 51	Y	N
Gray	102, 102, 102	Y	N
lightGray	153, 153, 153	Y	N
veryLightGray	192, 192, 192	N	Y
veryVeryLightGray	204, 204, 204	Y	N
veryDarkWhite	219, 219, 219	N	N
darkWhite	238, 238, 238	Y	N
white	255, 255, 255	Y	Y
veryDarkGreen	0, 102, 0	Y	N
darkGreen	0, 204, 0	Y	N
green	0, 255, 0	Y	Y
darkYellow	204, 204, 0	Y	N
yellow	255, 255, 0	Y	Y
darkBlue	0, 0, 204	Y	N
blue	0, 0, 255	Y	Y
lightBlue	0, 153, 255	Y	N
veryLightBlue	102, 153, 255	Y	N
veryVeryLightBlue	153, 204, 255	Y	N
darkCyan	0, 204, 255	Y	N
cyan	0, 255, 255	Y	Y
lightCyan	204, 255, 255	Y	N
purple	153, 0, 204	Y	N
lightPurple	153, 102, 204	Y	N

TABLE 59. **Color Names**

violet	201, 102, 255	Y	N
magenta	255, 0, 255	Y	Y
pink	255, 153, 255	Y	N
lightPink	255, 204, 255	Y	N
veryDarkBrown	204, 51, 0	Y	N
darkBrown	204, 102, 53	Y	N
brown	204, 153, 51	Y	N
lightBrown	255, 204, 153	Y	N
red	255, 0, 0	Y	Y
orange	255, 102, 51	Y	N