

# Grace Documentation: Using GUI Components in the objectdraw dialect in Grace

Kim B. Bruce

August 17, 2016

## 1 Introduction

Grace's objectdraw dialect supports GUI components modeled on those available in Java and Javascript.

Grace supports classes that generate GUI components of types Button, TextBox (for labels), TextField (for data entry), NumberField (for entering numbers), and Choice (for pop-up menus), all of which extend the type Component.

We begin by talking about these types and the classes that can be used to construct them. Later we will discuss how to place these components in a window.

## 2 GUI components

In this section we discuss the different kinds of GUI components supported by Grace and the methods that may be requested of them. However, first we take a brief detour to talk about events generated by components.

### 2.1 Events generated by GUI components

When the user interacts with a GUI event, the system generates an event that allows access to relevant information about the event, e.g., the component that was affected, the location of the mouse, etc. We list these types below:

```
type Event = {  
  // returns the component generating the event  
  source -> Component  
}  
  
type MouseEvent = Event & type {  
  // returns the mouse location when the event was generated  
  at -> Point  
}  
  
type KeyEvent = Event & type {  
  // returns the numeric code of the key pressed  
  code -> Number  
}
```

These events are generated by the operating system in response to user actions. Thus we do not need to worry about how to create them.

The event handling methods all take as parameter actions which the programmer can use to respond to the events. They all take events as parameters and then execute code. Their types are given below:

```
// type of a block that takes an event as parameter
```

```
type Response = {apply: Event → Done}
```

```
type MouseResponse = {apply: MouseEvent → Done}
```

```
type KeyResponse = {apply: KeyEvent → Done}
```

An element of type MouseResponse is a block that takes a MouseEvent as a parameter. For example,

```
{msEvent: MouseEvent → print "the event was at {msEvent.at}"}
```

is a block that takes a MouseEvent as a parameter and then prints out the location associated with the event.

## 2.2 Component

Type Component contains the methods available on all GUI components. While no classes generate objects with type exactly matching Component, all classes generating GUI components support all the method of Component:

```
type Component = {
```

```
// The width of this component.
```

```
width → Number
```

```
// The height of this component.
```

```
height → Number
```

```
// The dimensions of this component
```

```
size → Point
```

```
// Respond to a mouse click (press and release) in this component with the  
// given event.
```

```
onMouseClickedDo (f : MouseResponse) → Done
```

```
// Respond to a mouse press in this component with the given event.
```

```
onMousePressDo (f : MouseResponse) → Done
```

```
// Respond to a mouse release in this component with the given event.
```

```
onMouseReleaseDo (f : MouseResponse) → Done
```

```
// Respond to a mouse move in this component with the given event.
```

```
onMouseMoveDo (f : MouseResponse) → Done
```

```
// Respond to a mouse drag (move during press) in this component with the  
// given event.
```

```
onMouseDownDo (f : MouseResponse) → Done
```

```
// Respond to a mouse entering this component with the given event.
```

```
onMouseEnterDo(f : MouseResponse) → Done
```

```

// Respond to a mouse exiting this component with the given event.
onMouseExitDo (f : MouseResponse) -> Done

// Respond to a key type (press and release) in this component with the given
// event.
onKeyTypeDo (f : KeyResponse) -> Done

// Respond to a key press in this component with the given event.
onKeyPressDo (f : KeyResponse) -> Done

// Respond to a key release in this component with the given event.
onKeyReleaseDo (f : KeyResponse) -> Done

// Whether this component will dynamically fill up any empty space in the
// direction of its parent container.
isFlexible -> Boolean

// Set whether this component will dynamically fill up any empty space in the
// direction of its parent container.
flexible := (value : Boolean) -> Done

}

```

The mouse event handlers all take parameters of type `MouseResponse`, which is a block that takes a parameter of type `MouseEvent` (see the previous section).

## 2.3 TextBox

An object of type `TextBox` is a GUI component that can serve as a label for another component.

```

type TextBox = Component & type {

  // The text contents of the box.
  contents -> String
  contents := (value : String) -> Done
}

// create a label show the string in contents'
class textBoxWith (contents' : String) -> TextBox

```

As shown above, the class `textBoxWith` is used to create objects of type `TextBox`. Thus `textBoxWith ("Enter your name: ")` will create a label that reads “Enter your name”.

## 2.4 Button

An object of type `Button` is a labeled button that can be pressed to generate an event. Type `Labeled` is a synonym of `Button` and is included because other types will extend it.

```

type Labeled = Component & type {

  // The label on the button.
  label -> String
  label := (value : String) -> Done
}

```

```
type Button = Labeled
```

```
class buttonLabeled (label' : String) -> Button
```

A button saying “Press Me” can be generated by `buttonLabeled ("Press Me")`.

## 2.5 TextField and NumberField

The types `TextField` and `NumberField` generate components where users can add strings and numbers. They both extend the type `Input`.

```
type Input = labeled & type {
```

```
// Respond to this input gaining focus with the given action.
```

```
onFocusDo (f : Response) -> Done
```

```
// Respond to this input losing focus with the given action.
```

```
onBlurDo (f : Response) -> Done
```

```
// Respond to this input having its value changed (requires typing return).
```

```
onChangeDo (f : Response) -> Done
```

```
}
```

```
type TextField = Input & type {
```

```
// The contents of the text field.
```

```
text -> String
```

```
text := (value : String) -> Done
```

```
}
```

```
class textFieldLabeled (label' : String) -> TextField
```

```
type NumberField = Input & type {
```

```
// The contents of the number field.
```

```
number -> Number
```

```
number := (value : Number) -> Done
```

```
}
```

```
class numberFieldLabeled (label : String) -> NumberField
```

```
type Choice = Input & type {
```

```
// The currently selected option.
```

```
selected -> String
```

```
selected := (value : String) -> Done
```

```
}
```

```
class menuWithOptions (Iterable[[String]]) -> Choice
```

For example, evaluating `menuWithOptions ["one","two","three"]` creates a pop-up menu with three choices.

## 2.6 DrawingCanvas

An object of type DrawingCanvas is a rectangular portion of a window in which the programmer can draw graphic items of type Graphic, included filled and framed rectangles and ovals, as well as lines, text items, and images. These items are described more completely in other documents.

```
type DrawingCanvas = {  
  // Start drawing on the canvas. Will continue until the canvas is destroyed.  
  startDrawing -> Done  
  
  // add d to canvas  
  add(d: Graphic)->Done  
  
  // remove d from window  
  remove(d: Graphic)->Done  
  
  // tell the system to redraw the screen  
  notifyRedraw -> Done  
  
  // clear the canvas  
  clear -> Done  
  
  // Send d to top layer of graphics  
  sendToFront (d: Graphic)->Done  
  
  // send d to bottom layer of graphics  
  sendToBack (d: Graphic)->Done  
  
  // send d up one layer in graphics  
  sendForward (d: Graphic)->Done  
  
  // send d down one layer in graphics  
  sendBackward (d: Graphic)->Done  
  
  // return the current dimensions of the canvas  
  width -> Number  
  height -> Number  
  size -> Point  
}
```

```
class drawingCanvasSize (dimensions': Point) -> DrawingCanvas
```

## 3 Adding components to a window

Most programs using the objectdraw library will require a canvas in the middle of the window. Objects and classes that inherit graphicApplicationSize(dim) will have a canvas preinstalled, while other GUI components can be added above or below it. On the other hand, objects or classes that inherit applicationTitle (windowTitle) size (dim) will start with no GUI components in the window.

### 3.1 Application and GraphicApplication

Types Application and GraphicApplication both extend type Container.

```

// The type of components that contain other components.
type Container = Component & type {

    // The number of components inside this container.
    numComponents -> Number

    // Retrieve the component at the given index.
    at (index: Number) -> Component

    // Put the given component at the given index.
    at (index: Number) put(component: Component) -> Done

    // Add a component to the end of the container.
    append (component: Component) -> Done

    // Add a component to the start of the container.
    prepend (component: Component) -> Done

    // Perform an action for every component inside this container.
    do (f: Procedure[[Component]]) -> Done

    // Arrange the contents of this container along the horizontal axis.
    // Components which exceed the width of the container will wrap around.
    arrangeHorizontal -> Done

    // Arrange the contents of this container along the vertical axis.
    // Components which exceed the height of the container will wrap around.
    arrangeVertical -> Done
}

// A standalone window which contains other components.
type Application = Container & type {

    // The title of the application window.
    windowTitle -> String
    windowTitle := (value : String) -> Done

    // must be requested in order to pop up window when initialization complete
    startApplication -> Done

    // Close the window
    stopApplication -> Done
}

class applicationTitle(initialTitle : String)
    size (initialDimensions: Point) -> Application

// Type of object that runs a graphical application that draws
// objects on a canvas and responds to mouse actions on the canvas
type GraphicApplication = Application & type {
    // canvas holds graphic objects on screen
    canvas -> DrawingCanvas

```

```

// Respond to a mouse click (press and release) in the canvas at the given
// point.
onMouseClicked (mouse : Point) → Done

// Respond to a mouse press in the canvas at the given point.
onMousePress (mouse : Point) → Done

// Respond to a mouse release in the canvas at the given point.
onMouseRelease (mouse : Point) → Done

// Respond to a mouse move in the canvas at the given point.
onMouseMove (mouse : Point) → Done

// Respond to a mouse drag (move during press) in the canvas at the given
// point.
onMouseDown (mouse : Point) → Done

// Respond to a mouse entering the canvas at the given point.
onMouseEnter (mouse : Point) → Done

// Respond to a mouse exiting the canvas at the given point.
onMouseExit (mouse : Point) → Done

// must be invoked to create window and its contents as well as prepare the
// window to handle mouse events
startGraphics → Done
}

```

### 3.2 Adding Components

You can add components to an object of type `Application` or `GraphicApplication` by using the commands `prepend` and `append`. These methods add components either at the very beginning or at the very end of the list of components.

For example, suppose `drawingProgram` is an object of type `GraphicApplication`. Thus it starts out as having a canvas in the window. If `button1`, `button2`, and `menu` are GUI components then the results of executing:

```

prepend (button2)
prepend (button1)
append (menu)

```

will result in `button1` and `button2` being next to each other (in that order) above the canvas, while `menu` is centered below the canvas in the window.

You can have more control over the arrangement of the components by inserting them in containers and placing those containers in the window or in other containers. You can create a container by evaluating `emptyContainer`.

Here is an example of using containers to arrange the GUI components more carefully:

```

def southBox = emptyContainer
// items are laid out in southBox from top to bottom
southBox.arrangeVertical

// container row1 will hold boxLabel and sizeField
def row1 = emptyContainer

```



Figure 1: Layout of GUI components in a window

```

row1.append (boxLabel)
row1.append (sizeField)

// container row2 will hold colorLabel and colorMenu
def row2: Container = emptyContainer
row2.append (colorLabel)
row2.append (colorMenu)

// container row3 will hold three buttons
def row3 = emptyContainer
row3.append (slowButton)
row3.append (mediumButton)
row3.append (fastButton)

// Add the rows from top to bottom in southBox
southBox.append (row1)
southBox.append (row2)
southBox.append (row3)

// add the whole southBox to the bottom of the window, below the canvas
append (southBox)

```

The container `southBox` contains three other containers: `row1`, `row2`, and `row3`, which will be stacked one on top of the other (that is determined by `southBox.arrangeVertical`). Each of the row containers contains several GUI components, which are laid out next to each other, as shown in Figure 1.

The same idea works with objects inheriting from `application` except there will be no canvas in the window.

## 4 Responding to events involving GUI components

As can be seen in the type `Component`, GUI components can respond to user actions. For example, components can respond to mouse presses or clicks, selecting an item in a pop-up menu, or entering a new item in a text or number field.



## 4.1 Button events

An object of type `Button`, can respond to various mouse events. Normally, however, we will only have them respond to a press or click by a mouse. Suppose `tickleButton` has type `Button`. If we wish for it to respond to a mouse press by printing a message then we can write:

```
tickleButton.onMousePress {mEvt: MouseEvent -> print "Stop tickling me!"}
```

Now when the user presses the mouse on the button, it will print “Stop tickling me!”. The code after the arrow can use the formal parameter `mEvt`, for example by sending it method requests for `source` or `at`. However, it will be unusual to need those pieces of information (especially the location of the mouse in the button) in the code reacting to the press.

If you prefer to react to a mouse click or release instead, you can use the corresponding method. You can add as many actions as you like to respond to mouse events. They will all be executed when the corresponding event occurs.

## 4.2 Input events

An object of a type extending `Input`, e.g., `Choice`, `TextField`, or `NumberField` will generate a `Change` event when a new item is selected from a pop-up menu or a new entry is made to a field (terminated by hitting the **enter** key).

These are handled using the `onChangeDo` method of the component.

```
menu.onChangeDo {evt: Event -> ... menu.selected ...}  
myTextField.onChangeDo {evt: Event -> ... myTextField.text ...}  
myNumberField.onChangeDo {evt: Event -> ... myNumberField.number ...}
```

As you can see, once the event is triggered we can query the item to see what value has been provided.