*This* **event** *selects the given items. This can be done by name or index. Vectors are accepted.*

---

**listboxcontainer.selection - Function**

| | |
|---|---|
| **Package** | **display** |
| **Module** | **widgets** |
| **Tool** | **listboxcontainer** |

*(EVENT) return a record of items selected*

**Synopsis**
*selection()*

**Returns**
*record*

**Description**
*The items selected in the* `listbox` *are returned as a record.*

---

---

### 1.4.20 multiautogui - Tool

| | |
|---|---|
| **Package** | **display** |
| **Module** | **widgets** |

*Tool to manage a number of autoguis*
*include "multiautogui.g"*

**Description**

*This* `tool` *is designed for use when context sensitive menus are required. For example, it was implemented for use with annotations, since clicking on a new shape should bring up options for that shape. This allows different "types" of autoguis to be built, one for each type of shape. The autoguis are built only once, since construction of them takes a significant amount of time. Once built they are mapped, and unmapped from the screen as needed. This widget also attempts to make accessing multiple autoguis simpler, by allowing "nested" options within the* *records passed to / from it. For example, you can construct a* `multiautogui`*, and have it generate multiple autoguis, all with their parameters filled, by forming the* `fillgui` *record correctly. Most of the work is still done by the underlying autoguis.*

**Parameter description** *In a similar manner to autoguis, most transactions*

*occur via "parameter records". In addition to this are control functions, which control the mapping to and from the screen of the different autoguis. The actual records destined for the desired autogui are nested one level down. For example, if two autoguis are desired, "someoptions" and "otheroptions" and you wish them both to occupy the same spot on screen using a multiautogui, you would create a parameter record as follows, construct the widget, and then choose which one you wanted visible:*

- `myoptionrecord:`
  - `someoptions:` *This is the name you will use to refer to operations on this autogui, and also the name which the multiautogui will return new options for this set under*
    - `options:` *A standard option set for an autogui*
  - `otheroptions:` *This is just some other name.*
    - `options:` *Another different standard option set for an autogui*

```
include 'multiautogui.g'

mymulti := multiautogui(myoptionrecord, title = 'Test Multi');

mymulti.show('otheroptions');
```

*For a complete example, see multiautoguitest in multiautogui.g . A similar technique is used to fill the GUIs, and also to receive options from them.* **Event**

**description**

*Basically, anytime one of the autoguis being maintained by multiautogui fires a* `setoptions` *event, it is passed on by multiautogui. The difference is multiautogui stores the parameters returned by the autogui in a record which has the name of the parameter set. The example above, in which we created two autoguis called "someoptions" and "otheroptions" provides a good way to demostrate this. We imagine that "someoptions" was visible, and the user changed a parameter. Multiautogui would emit a* `setoptions` *event, which contained a record with one field. The field name would be the originating autogui (in this case "someoptions"), and contained within that subrecord would be the new options. Similarly, when a* `newuserchoicelist` *is emitted (in fact, all events (*) are forwaded on in this manner), the same "parameter encoding" occurs, with the option record being stored in a subrecord, the name of which is determined by the originating autogui.*

**Example**
*For a simple example, please refer to multiautogui.g. It contains a subsequence "multiautoguitest" which demonstrates multiautogui managing three autoguis.*

### multiautogui - Constructor

| | |
|---|---|
| **Package** | **display** |
| **Module** | **widgets** |
| **Tool** | **multiautogui** |

*Create a multiautogui* `tool`

**Synopsis**
*multiautogui(params, title, toplevel, actionlabel, autoapply, expand, widgetset)*

**Arguments**

| | |
|---|---|
| *params* | *Record to define multiple autoguis*<br>*Allowed: Record* |
| *title* | *title string if top-level frame constructed internally*<br>*Allowed: String*<br>*Default: 'autogui'* |
| *toplevel* | *top-level parent frame*<br>*Allowed: frame or boolean*<br>*Default: F* |
| *actionlabel* | *label for action button (passed to each autogui) TODO: It would be nice to have a common "action" button*<br>*Allowed: string or boolean*<br>*Default: F* |
| *autoapply* | *Show auto-apply buttons*<br>*Allowed: boolean*<br>*Default: T* |
| *expand* | *Expand policy*<br>*Allowed: 'x'—'y'—'both'—'none'*<br>*Default: 'none'* |
| *widgetset* | *widgetserver* `tool` *to use to build the GUI*<br>*Allowed: widgetserver* `tool`<br>*Default: dws* |

**Returns**
*subsequence*

**Description**

*This is the only constructor for the multiautogui* `tool`*. The first argument,* `params`*, contains the various parameters which will be used to generate the multiple GUIs. If a parent frame is not given in the* `toplevel` *argument, a top-level frame will be created with title as given in the* `title` *argument. If the* `actionlabel` *argument is given as a string, then a button will be placed at the bottom of each of the GUIs with that string as its label. Pressing this button will cause the autogui* `tool` *to emit all parameters and their values via a* `setoptions` *event. The final two arguments just provide some future flexibility with respect to widgetserver and guientry* `tools`*.*

---

### multiautogui.shownone - Function

| | |
|---|---|
| **Package** | **display** |
| **Module** | **widgets** |
| **Tool** | **multiautogui** |

*Show no autoguis in the frame (unmap all)*

**Synopsis**
*shownone()*

**Returns**
*T*

**Description**
*This function unmaps all autoguis, so none are visible.*

---

### multiautogui.show - Function

| | |
|---|---|
| **Package** | **display** |
| **Module** | **widgets** |
| **Tool** | **multiautogui** |

*Read the GUI and return the current settings*

**Synopsis**
*show(which)*

### Arguments

| | |
|---|---|
| *which* | *which autogui to show*<br>*Allowed: String - name of a created autogui* |

### Returns
*T if gui name found, else F*

### Description

*This function allows one of the autoguis to be shown. The name supplied here should be the name of one of the created autoguis.*

---

## multiautogui.fillgui - Function

| | |
|---|---|
| **Package** | **display** |
| **Module** | **widgets** |
| **Tool** | **multiautogui** |

*Fill one or all of the autoguis*

### Synopsis
*fillgui(options)*

### Arguments

| | |
|---|---|
| *options* | *options to fill autgui(s)*<br>*Allowed: Record to use to fill autogui(s)* |

### Returns
*T*

### Description
*This function is used to fill the autoguis. The parameter must be a record whose field names correspond to one or more autoguis. Within these fields should be the options with which to fill the respective autogui. If one of the field names does not exist as an autogui, its contents are used to create a new one.*

---

## multiautogui.addautogui - Function

| | |
|---|---|
| **Package** | **display** |
| **Module** | **widgets** |
| **Tool** | **multiautogui** |

*Add another autogui to the widget*

**Synopsis**
*addautogui(name, params)*

**Arguments**

| | |
|---|---|
| *name* | *name of the new autogui*<br>*Allowed: String* |
| *params* | *parameter to fill autogui*<br>*Allowed: Record with autogui information* |

**Returns**
*T if addition successful, else F*

**Description**

*This function can be used to add a new autogui to the collection. It takes a name (which will from then on be used as an identifier in all option records) and a set of parameters which will be used to construct the new autogui.*

---

### multiautogui.multiautoguitest - Function

| | |
|---|---|
| **Package** | **display** |
| **Module** | **widgets** |
| **Tool** | **multiautogui** |

*Test the multiautogui*

**Synopsis**
*multiautoguitest()*

**Returns**