# Using fields

November 28, 2022

# Contents

**Defining a field** **3**
Paths to fields . . . 4
Path segments . . . 6
Special and advanced techniques . . . 7
Optical fields . . . 11
Using the built-in screenshot-taker . . . 11
Testing a path . . . 13
Using Cuesta . . . 13
Using a flow or the debugger . . . 13

**Fields API** **14**
Click . . . 14
Click with offset . . . 15
Simulated Click . . . 15
Simulated click with offset . . . 16
Right click . . . 16
Right-click with offset . . . 16
Double click . . . 17
Double-click with offset . . . 18
Click cell . . . 18
Read . . . 19
Bounds . . . 20
Exists . . . 20
Inspect . . . 21
Reflection depth . . . 21
Find / FindAll . . . 22
Input . . . 23
Native input . . . 24
Native input with delay . . . 24
Select . . . 25
Select with index . . . 26
Select with offset . . . 27
Select with offset and skip . . . 27
Toggle a checkbox/radiobutton . . . 28
Check/Uncheck . . . 28
isChecked . . . 28

Edit cell . . . . . 28
Highlight . . . . . 30
Highlight with color . . . . . 30
Lowlight . . . . . 30
Eval . . . . . 30

When configuring an application for automation purposes it is often necessary to interact with the user-interface of the application in some manner. A *field* as concept in Cuesta represents an element in the user-interface which can be interacted with.

This can be a button, a dropdown, a table or any other type of user-interface element. Once defined a field can be manipulated in a flow, e.g. clicking a button named `Ok` would look like the following in a flow:

```
1 Fields.Ok.click();
```

What happens in that statement is that we get the `Ok` field from the `Field` object. If the field name is not a valid Javascript variable name, then use the object indexing scheme instead, e.g.:

```
1 Fields['Ok'].click();
```

## Defining a field

A field can be identified from its *path* or using a *screenshot* of the field. The path approach utilizes structural information in the user-interface while the screenshot is purely visual making it more brittle wrt changes in application appearance. The Cuesta form for defining a field is given below:

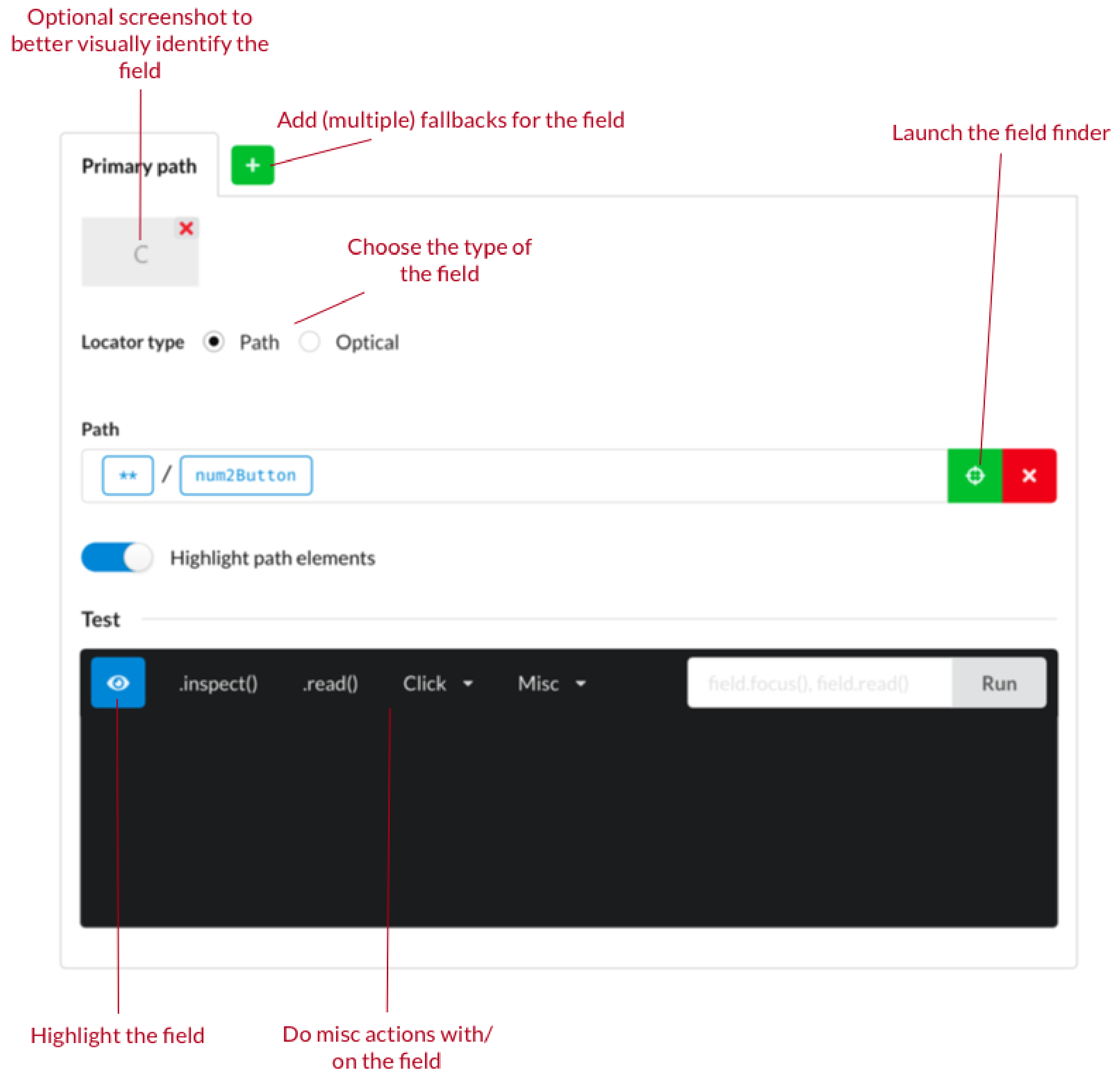
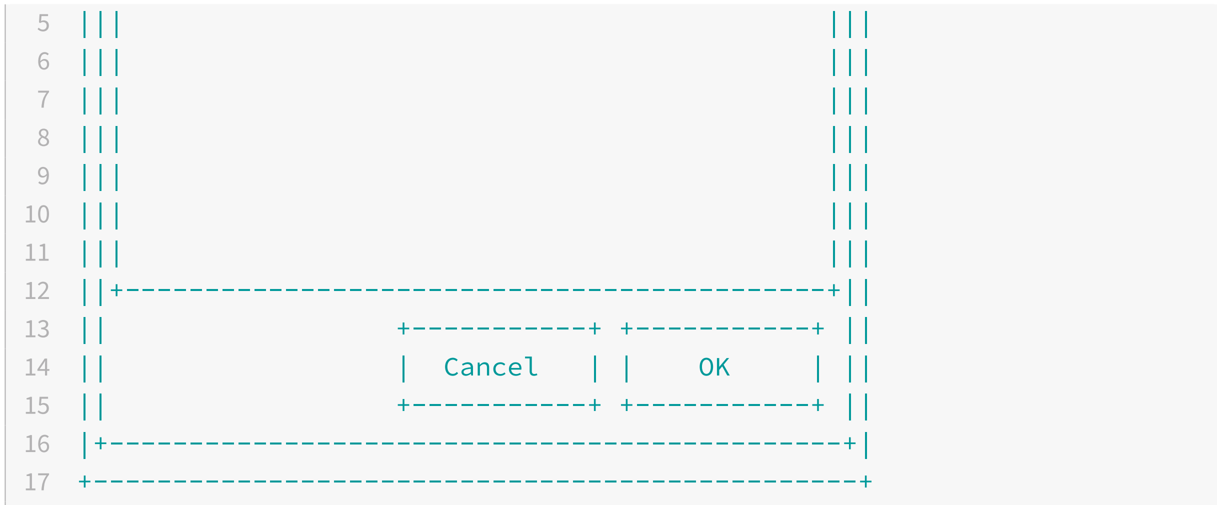



**Figure 1:** Defining a field in Cuesta

### Paths to fields

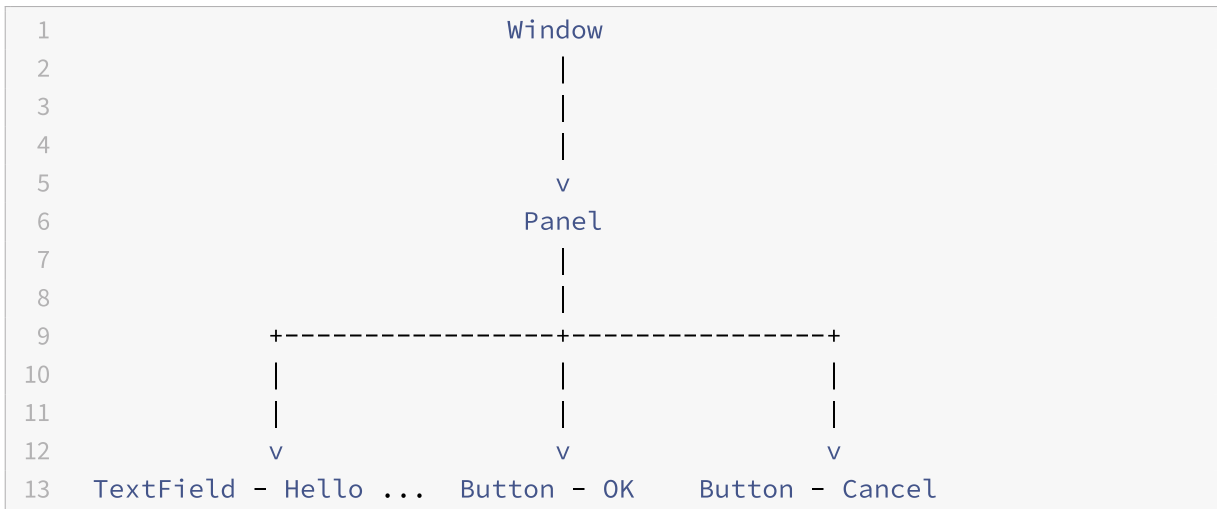
A user-interface has a structure like a tree with the root of the tree being the window and the elements the branching structure. For instance the following application layout:

```
+-----------------------------------------------+
|+---------------------------------------------+|
||+-------------------------------------------+||
|||Hello, I'm a text-field.                   |||
```



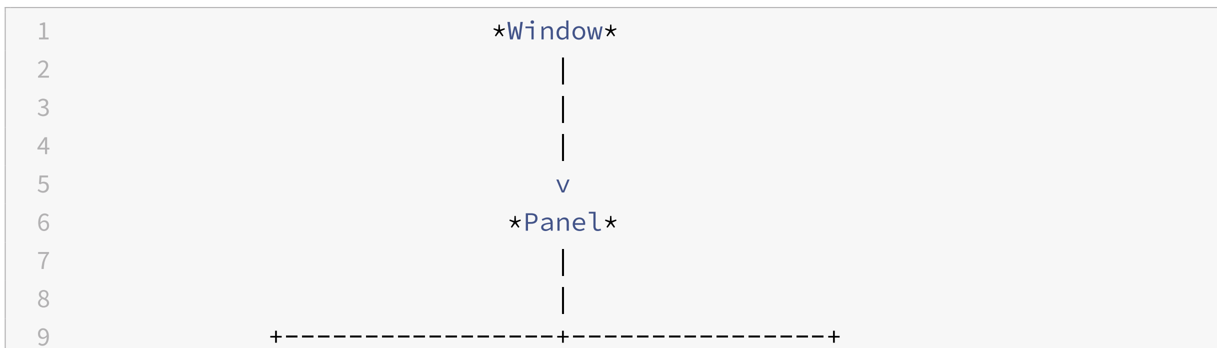
```
5   |||                                            |||
6   |||                                            |||
7   |||                                            |||
8   |||                                            |||
9   |||                                            |||
10  |||                                            |||
11  |||                                            |||
12  ||+--------------------------------------------+||
13  ||                 +------------+ +------------+ ||
14  ||                 |  Cancel    | |     OK     | ||
15  ||                 +------------+ +------------+ ||
16  |+----------------------------------------------+|
17  +------------------------------------------------+
```

The structure of the user-interface above can be mapped to a tree like so:



```
1                            Window
2                              |
3                              |
4                              |
5                              v
6                            Panel
7                              |
8                              |
9              +---------------+---------------+
10             |               |               |
11             |               |               |
12             v               v               v
13  TextField - Hello ...   Button - OK    Button - Cancel
```

To identify e.g. the *Cancel* button we use a scheme where you provide the *path* from the root of the application to the element to be identified. In the example above an identifying could be (marked with $\star$):



```
1                           *Window*
2                              |
3                              |
4                              |
5                              v
6                           *Panel*
7                              |
8                              |
9              +---------------+---------------+
```

```
10               |                  |                 |
11               |                  |                 |
12               v                  v                 v
13   TextField - Hello ...  Button - OK    *Button - Cancel*
```

And in a textual form this translates to:

```
1  Panel/Button - Cancel
```

#### Path segments

Paths are compromised of a number of segments; one for each step down in the tree structure. Each segment matches itself against a user-interface element, checking a set of predefined properties on the element. These are commonly:

- The *type* of the element (e.g. `TextField`, `Label`)
- The *automation-id* if present
- The *textual content* of the element
- The given *name* of the element
- and more …

Thus the path in the previous example can be shortened to:

```
1  Panel/Cancel
```

The default matching algorithm for each segment simply checks if the given string is a substring of the extracted element property. Using a few simple operatorswe provide more flexibility and greater accuracy:

- `*ancel` matches any string ending with "ancel"
- `Can*` matches any string starting with "Can"
- `*an*` matches any string containing "an"
- `^.+an.+^` matches the string against the regular expression `.+an.+`

Furthermore the segment `**` can be used to match deep into the structure, e.g. the path:

```
1  **/Cancel
```

Will match the first element matching `Cancel` anywhere in the structure. This can also be used like so:

```
1  Panel/**/Cancel
```

Which matches any `Cancel` element with a `Panel` ancestor.

### Special and advanced techniques

The following is a list of semi-rarely used techniques which can prove useful in tricky and non-accessible user-interfaces.

### Locating via relative position

If it is not possible to locate a field directly then it may be possible to use another more easily locatable field as a sort of anchor and then using the relative (according to the anchor) position of the desired field as a guide. This is often the case with fields that have easily locatable *labels*. Using the *label* as the anchor and then specifying e.g. that the desired field is the textfield below the anchor is then possible. A positional path looks like:

```
1 **/Panel/AnchorElement<above>DesiredElement
```

This path will cause the `AnchorElement` to be found and then a search for the nearest `DesiredElement` **below** the anchor. It should be read as “look for a `Panel` element somewhere in the tree, then find a child `AnchorElement` which is positioned *above* a `DesiredElement` which is our target”.

The possible positional hints are:

- `<above>` the anchor element must be found above the target
- `<below>` the anchor element must be found below the target
- `<left-of>` the anchor element must be found left of the target
- `<right-of>` the anchor element must be found right of the target
- `<nearest>` the nearest matching element to the anchor is selected

The *closest* match is the one selected if there are more matching elements in all the above cases.

Another possibility is to define a field based on the path to an element and an offset in pixels from that element to find a second element. The syntax looks as follows: `**/OkButton<offset-with>@-10|200`. This will locate an element 10 pixels to the left and 200 pixels below the upper left corner of some Ok button.

### Locating via structural hints

For fields that can best be identified by their placement in the UI tree, we can use structural hints. These allow an element to be identified by a unique child or sibling it may have. An example could be an input with no unique features, which has an easily identifiable label as its sibling. The path syntax is like the positional hints:

```
**/Panel/AnchorElement<sibling-of>DesiredElement
```

The available structural hints are:

- `<child-of>` the anchor element must be a child of the target
- `<sibling-of>` the anchor element must be a sibling of the target

#### Skipping matches

In case the user-interface contains "twins" i.e. undistinguishable elements in the same level of the tree then then the *skip* operator (`#`) can be used to select the *i'th* matching sibling. Consider the following tree:

```
Panel
  |- Button
  |- Button
  `- Button
```

In order to target the 2nd button we might use the following path:

```
Panel/Button#1
```

The skip operator can only be used with the last element in the path and will thus only apply to the siblings of the targeted element.

#### Restricting path property types (native applications only)

To increase path resolution speed in native applications you can specify which property on the UI element should be used for matching each path segment by prefixing the segment with `(<type-goes-here>)`. A button with the text "Ok" can then be specified with `**/(text)Ok`. The type can appear on all segments e.g.

```
**/(type)Panel/(text)Ok
```

Which resolves any UI element with the text "Ok" directly inside a panel.

The available property-types are:

- `id` the (automation)id of the element
- `text` the text representation of the element (normally the text you can see in the UI)
- `class` the class of the UI element
- `type` the type of the UI element

- `name` the name of the UI element

The `type` can be "button", "panel", "menu", "textbox" etc.

#### Targeting a native menu

Paths prefixed with `@` traverse the menu of an application. E.g.

```
1 new Field("@Files/Save").click();
```

will try to open the "Files" menu, then click the "Save" option.

#### Backtracking (in web-applications only)

Another rarely occurring case is when the target can only be uniquely matched by targeting one of its descendents. In the following example we have a clearly locatable *Ok* `Button` but we are really only interested in an anonymous `Panel` locate two levels up in the tree.

```
1 Panel                  <- this is our target
2   `- Panel
3        `- Button - Ok  <- this is the Button we can locate
```

Here we can target the desired panel by way of the following path:

```
1 [Panel]/Panel/Ok
```

The `[` and `]` effectively tells the path targeting mechanism to do the full path resolution but return the element contained within.

#### Using CSS selectors (in web-applications only)

An alternative path format for use in web-applications is using CSS selectors. In some cases using CSS selectors is easier and faster. E.g. for finding an element with a specific id:

```
1 #the-id
```

vs the normal path format:

```
1 **/the-id
```

The latter being faster.

### Searching a specific embedded window

Given a multi-window application or an application with many embedded "windows", it is sometimes useful to limit the search for a given element to a specific window. This is done by prefixing the path with `{title-of-window}` and thus limiting the search to any windows whose title matches the given. E.g.:

```
1 {MyWindow}**/Panel/Ok
```

### Searching in all windows on the desktop

For native apps, it is possible to break out of the current application when searching for a window by prepending the window-matching part of the field with `!`.

```
1 {!Foo}**/Ok
```

Will search all windows matching `Foo` and return the first match on the rest of the path.

For other app types, the exclamation mark in the window matcher causes manatee to use a native driver to traverse the indicated window. This is useful eg for native dialogs in a chrome browser, which are otherwise difficult to interact with.

In a chrome app, click a dialog ok button with a path like this:

```
1 {!}**/OK
```

The empty window matcher is shorthand for traversing the main window of the chrome app. To access a native calculator from a chrome app, use a path like this:

```
1 {!Calculator}**/ResultPane
```

### Matching invisible controls

For java and wep apps, it is occasionally possible and desirable to interact with controls that are not visible to the user.

Note that in doing so, you are straying from what the developers of the application intended. As a consequence, the application may behave unexpectedly. Thorough testing is adviced.

By default, invisible path matches are inaccessible for automation. You can however indicate to manatee that you want to allow such matches with an exclamation mark at the end of the path like this:

```
1 **/HiddenButton!
```

Such a path will match the `HiddenButton` control even if it isn't visible.

```
1  **/Ok#2!
```

This path finds two sibling `Ok`-buttons (or similar) and matches the second one - even if it isn't visible.

Determining the presence of invisible controls can be tricky. They can sometimes be found with the field finder while they are visible and testing may then reveal that the path still works when the control is hidden (when an exclamation mark is added as illustrated above).

Another way to find hidden UI controls is by analyzing application structure with the `inspect` command. The inspect output includes a boolean `visible` property indicating whether an exclamation mark is required to match the control.

## Optical fields

Optical fields are simply small screenshots of the user-interface element with an optional offset which Manatee will try to find visually and translate to a proper element. The offset is used e.g. when clicking s.t. the actual click is offset from the found location of the element.

### Using the built-in screenshot-taker

If the field can only be identified by a screenshot press the Grab screenshot button. A red square will appear which can be move and resized to fit the field. When the red square fits the field click on the square once. The square turns green and is now fixed to the field. In order to be able to click on the field (if applicable) click on the position on the screen where the click must be done. A red dot will show where the click will be done.

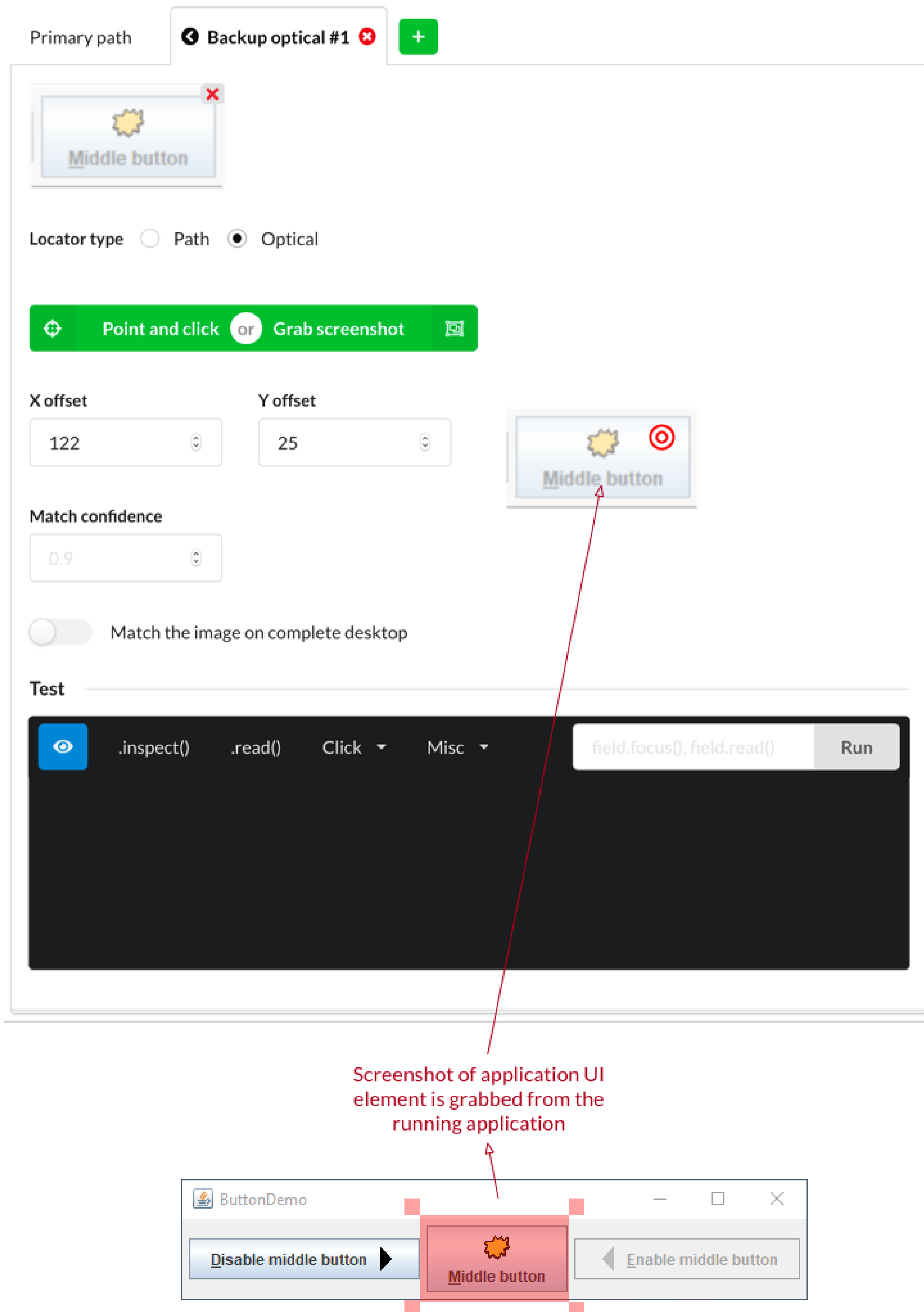



**Figure 2:** The built-in screenshot taker in action

The screenshot is shown in Cuesta. The click position can be adjusted in the X and Y offset fields. Match confidence can be set to reduce how acurately the screenshot should match the graphics on the screen in the application in order to have a match on the field. It can typically be set to 0.7.

### Testing a path

Given a path it is useful to be able to see that the element found when the path resolution is done in Manatee is the correct element found. This can be done directly from Cuesta or by using the field in a flow.

#### Using Cuesta

Activating the locate button in Cuesta will cause a local Manatee to *highlight* the field found. This is a quick and easy way to check whether the path is correct or not.

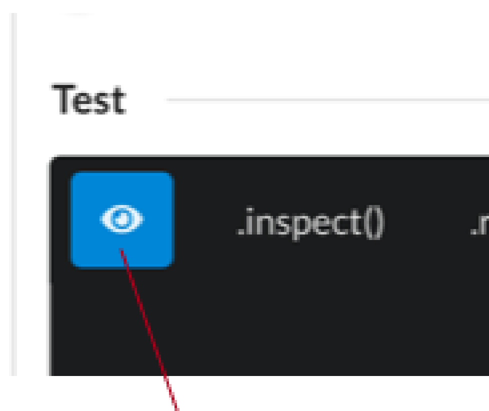



**Figure 3:** Click the locate button to find and highlight the field

#### Using a flow or the debugger

It is also possible to use a flow or the REPL in the `Debug.ger()` to *highlight*, *inspect* or otherwise manipulate and test a path. Fields can be created on-the-fly in a flow meaning that the following

code is a quick way to try out a path:

```
var f = new Field('**/Panel/Ok');
f.highlight(); // to try and highlight the element
f.click(); // to try and click the element
f.inspect(); // to gain more info about the element
// and other field methods
```

## Fields API

Once a field has been defined it can be used in a flow. Depending on the type of field (i.e. whether it represents a button, a panel or something else) the following methods are available.

### Click

Will click on the given field.

#### Parameters

- `options` an optional options object, supports;
  - `deadline` the time in ms to wait for the click to fail/succeed. If the click takes longer than the deadline to fail or succeed it will be reported as succeeding to the caller. The use-case for the `deadline` parameter is for example if the click launches a dialog which blocks the thread, then setting a deadline allows the flow to continue even though the click is technically not done.
  - `useCachedUI` boolean indicating if UI component lookup should use the UI itself or the underlying model. Defaults to **`false`** (underlying model traversal).
  - `useFieldCache` boolean indicating whether to use a cache to lookup the field - may be significantly faster in some cases - default **`false`**
  - `fieldCacheExpiry` int indicating the useable age in seconds of the field retrieved from the cache (only relevant if `useFieldCache` is **`true`**) - default is configurable as a global option

#### Example

```
Fields["mybutton"].click();

// With an optional 500 ms deadline and try to retrieve the field from
   the cached fields
```

```
4 Fiels["mybutton"].click({ deadline: 500, useFieldCache: true });
```

### Click with offset

Will click on the given field offset by the amount given. It allows you to e.g. click in the middle of a table row or the corner of a button.

#### Parameters

- `x` the number of pixels from the *left* of the element to click
- `y` the number of pixels from the *top* of the element to click
- `options` an optional options object, supports;
  - `deadline` the time in ms to wait for the click to fail/succeed. If the click takes longer than the deadline to fail or succeed it will be reported as succeeding to the caller. The use-case for the `deadline` parameter is for example if the click launches a dialog which blocks the thread, then setting a deadline allows the flow to continue even though the click is technically not done.
  - `useFieldCache` boolean indicating whether to use a cache to lookup the field - may be significantly faster in some cases - default **`false`**
  - `fieldCacheExpiry` int indicating the useable age in seconds of the field retrieved from the cache (only relevant if `useFieldCache` is **`true`**) - default is configurable as a global option

#### Example

```
1 // Click myButton 20px from left and 10px from top
2 Fields["mybutton"].clickWithOffset(20, 10);
```

### Simulated Click

Will simulate a mouse-click on the given field. The difference between simulate-click and click is only relevant for Java applications where mouse-events can be generated directly (click) or as a series of injected events - mousedown, mouseclicked, mouseup (simulateClick).

#### Example

```
1 Fields["mybutton"].simulateClick();
```

### Simulated click with offset

Will click on the given field offset by the amount given. It allows you to e.g. click in the middle of a table row or the corner of a button.

#### Example

```
1 // Click myButton 20px from left and 10px from top
2 Fields["mybutton"].simulateClickWithOffset(20, 10);
```

### Right click

Will right-click on the given field.

#### Parameters

- `x` the number of pixels from the *left* of the element to click
- `y` the number of pixels from the *top* of the element to click
- `options` an optional options object, supports;
  - `deadline` the time in ms to wait for the click to fail/succeed. If the click takes longer than the deadline to fail or succeed it will be reported as succeeding to the caller. The use-case for the `deadline` parameter is for example if the click launches a dialog which blocks the thread, then setting a deadline allows the flow to continue even though the click is technically not done.
  - `useFieldCache` boolean indicating whether to use a cache to lookup the field - may be significantly faster in some cases - default **`false`**
  - `fieldCacheExpiry` int indicating the useable age in seconds of the field retrieved from the cache (only relevant if `useFieldCache` is **`true`**) - default is configurable as a global option

#### Example

```
1 Fields["mybutton"].rightClick();
```

### Right-click with offset

Will click on the given field offset by the amount given. It allows you to e.g. click in the middle of a table row or the corner of a button.

#### Parameters

- `options` an optional options object, supports;
  - `deadline` the time in ms to wait for the click to fail/succeed. If the click takes longer than the deadline to fail or succeed it will be reported as succeeding to the caller. The use-case for the `deadline` parameter is for example if the click launches a dialog which blocks the thread, then setting a deadline allows the flow to continue even though the click is technically not done.
  - `useFieldCache` boolean indicating whether to use a cache to lookup the field - may be significantly faster in some cases - default `false`
  - `fieldCacheExpiry` int indicating the useable age in seconds of the field retrieved from the cache (only relevant if `useFieldCache` is `true`) - default is configurable as a global option

#### Example

```
1 // Click myButton 20px from left and 10px from top
2 Fields["mybutton"].rightClickWithOffset(20, 10);
```

### Double click

Will double-click on the given field.

#### Parameters

- `options` an optional options object, supports;
  - `deadline` the time in ms to wait for the click to fail/succeed. If the click takes longer than the deadline to fail or succeed it will be reported as succeeding to the caller. The use-case for the `deadline` parameter is for example if the click launches a dialog which blocks the thread, then setting a deadline allows the flow to continue even though the click is technically not done.
  - `useFieldCache` boolean indicating whether to use a cache to lookup the field - may be significantly faster in some cases - default `false`
  - `fieldCacheExpiry` int indicating the useable age in seconds of the field retrieved from the cache (only relevant if `useFieldCache` is `true`) - default is configurable as a global option

#### Example

```
1  Fields["mybutton"].doubleClick();
```

### Double-click with offset

Will click on the given field offset by the amount given. It allows you to e.g. click in the middle of a table row or the corner of a button.

#### Parameters

- `x` the number of pixels from the *left* of the element to click
- `y` the number of pixels from the *top* of the element to click
- `options` an optional options object, supports;
  - `deadline` the time in ms to wait for the click to fail/succeed. If the click takes longer than the deadline to fail or succeed it will be reported as succeeding to the caller. The use-case for the `deadline` parameter is for example if the click launches a dialog which blocks the thread, then setting a deadline allows the flow to continue even though the click is technically not done.
  - `useFieldCache` boolean indicating whether to use a cache to lookup the field - may be significantly faster in some cases - default **`false`**
  - `fieldCacheExpiry` int indicating the useable age in seconds of the field retrieved from the cache (only relevant if `useFieldCache` is **`true`**) - default is configurable as a global option

#### Example

```
1  // Click myButton 20px from left and 10px from top
2  Fields["mybutton"].doubleClickWithOffset(20, 10);
```

### Click cell

Click in a cell in table (only applicable for tables). Clicking a cell has the following variants:

- `clickCell(...)` left-click a cell,
- `rightClickCell(...)`, and
- `doubleClickCell(...)`.

All with the following parameters:

#### Parameters

- `rowMatch` a text to match in the row - if an `integer` is supplied then that is used to select the row index
- `colMatch` a text to match in a column header - also use an `integer` here to use the column with that index
- `options` an options object on which the follow properties can be set;
  - `deadline` the time in ms to wait for the click to fail/succeed. If the click takes longer than the deadline to fail or succeed it will be reported as succeeding to the caller.
  - `reflectionDepth` indicates how deep to do the search for the `rowMatch` value (also see Reflection depth)
  - `useCachedUI` boolean indicating if UI component lookup should use the UI itself or the underlying model. Defaults to **`false`** (underlying model traversal).
  - `useFieldCache` boolean indicating whether to use a cache to lookup the field - may be significantly faster in some cases - default **`false`**
  - `fieldCacheExpiry` int indicating the useable age in seconds of the field retrieved from the cache (only relevant if `useFieldCache` is **`true`**) - default is configurable as a global option
  - `offsetX` an int denoting the offset to use for the click from the left-most border of the cell
  - `offsetY` an int denoting the offset to use for the click from the top-most border of the cell

#### Example

```
1 // Click in the cell defined by its row containing 'A' and its column (
     header) containing 'B'
2 Fields["myTable"].clickCell('A', 'B');
3 // The same command but use reflection depth to do a deeper search
4 Fields["myTable"].clickCell('A', 'B', { reflectionDepth: 2 });
5 // Click row 10 in column with header 'B'
6 Fields["myTable"].clickCell(10, 'B', { reflectionDepth: 2 });
7 // Click row 10 in column 1
8 Fields["myTable"].clickCell(10, 1, { reflectionDepth: 2 });
```

### Read

Will read the value of the field. Depending on the type of the field the behavior will differ, e.g. on a label it will return the text content of the label, for a text-field it will return the contents of the text-field. For a more complex container type it will return a JSON representation of the control (which

can be natively accessed in the flow as an object). See JSON serialisation for details on how different types are serialised.

#### Parameters

- `options` an optional options object with details regarding the inspection.
  - `useCachedUI` boolean indicating if UI component lookup should use the UI itself or the underlying model. Defaults to **`false`** (underlying model traversal).
  - `useFieldCache` boolean indicating whether to use a cache to lookup the field - may be significantly faster in some cases - default **`false`**
  - **`throws`** boolean indicating whether to throw an error if the field cannot be found. Defaults to true. Native, java and IE drivers do *not* support this option. They always throw when the field cannot be found.
  - `fieldCacheExpiry` int indicating the useable age in seconds of the field retrieved from the cache (only relevant if `useFieldCache` is **`true`**) - default is configurable as a global option

#### Example

```
var contents = Fields["mytextfield"].read();
```

### Bounds

This can be used to get the bounds (location and size) of the field.

```
var b = Field["OK"].bounds();
// b is now an object e.g.
// { width: 100, height: 100, x: 10, y: }
```

### Exists

Returns true if the field could be found.

#### Example

```
if(Fields["mytextfield"].exists()) {
  ...
}
```

### Inspect

Inspect a given field. The returned object will contain misc information about the field - the type of information depends on the type of the field.

The resulting object can additionally be used for finding and interacting with other UI elements inside the inspected field. Each object in the resulting object hierarchy provides the full palette of field operations.

#### Reflection depth

You can optionally obtain more detailed information about the data in eg treeviews. To do this, pass a positive `reflectionDepth` value as shown in the examples below.

As an example, reflectionDepth: 3 means the result includes fields such as `arrival.date.day` (3 steps) but not eg `patient.eyes.left.tla` (4 steps).

The `reflectionDepth` paramater affects the data available in the output under the objects in the control in question (eg treeview nodes). The main use of this feature is to determine which patterns to use with `Field['field'].select()` when simply selecting the rendered text doesn't work.

#### Parameters

- `options` an optional options object with details regarding the inspection.
  - `useCachedUI` boolean indicating if UI component lookup should use the UI itself or the underlying model. Defaults to **`false`** (underlying model traversal).
  - `includeChildren` boolean indicating if the children of the targeted element should be included in the result. Defaults to **`true`**. Set to false when targeting a high level container as the result may otherwise be a bit unwieldy.
  - `reflectionDepth` (see below)
  - `useFieldCache` boolean indicating whether to use a cache to lookup the field - may be significantly faster in some cases - default **`false`**
  - `fieldCacheExpiry` int indicating the useable age in seconds of the field retrieved from the cache (only relevant if `useFieldCache` is **`true`**) - default is configurable as a global option
  - `collectTexts` (Java only) tries to figure out text contents of rendered controls. May be useful in combination with `useCachedUI` in some tables which uses buffered renders.

#### Example

```
var info = Fields["mytextfield"].inspect();
// See which information was returned
Debug.showDialog(JSON.stringify(info));
// If info has a `text` property, then this will show the text
Debug.showDialog(info.text);

var detailedInfo = Fields["myTreeView"].inspect({ reflectionDepth: 2});
// This object includes extra data under the nodes of 'myTreeView'.
Debug.showDialog(JSON.stringify(detailedInfo));
```

#### Find / FindAll

On objects returned by `inspect`, you are provided the convenience methods `find` and `findAll`, which make it simpler to traverse the inspect result object structure. Where `find` returns the first match to your query, `findAll` returns them all. As a convenience, `find` and `findAll` are also made available directly on the field, so you don't have to call `inspect` explicitly. Manatee will issue an `inspect` command on your behalf and traverse the result according to your query.

#### Parameters

The parameter signatures of `find` and `findAll` are identical. * First argument is either - A string: Manatee will return the first object in the inspect result with a property value that exactly matches this string - A regular expression: Manatee will similarly scan all properties on all objects in the structure for one matching the regexp - A function: Manatee will call this function, passing in as the only argument each object in the structure one at the time. When the function returns true, that object is included in the `find`/`findAll` result. * Second argument is an options object which serves to modify the search algorithm and is also passed to `inspect` allowing its behavior to also be customized. The properties supported by `find`/`findAll` are as follows: - `skip` (number): Skip the indicated number of matches before any matches are accepted - `property` (string): Specify which properties to match. Only useful if the first argument is either a string or a regexp.

#### Custom traversal

If you need to traverse the structure in a way that can't be achieved with the `find`/`findAll` methods, you can create your own recursive traversal function. For each node in the structure, the `children` property provides access to the children of the node. See example below.

#### Example

```
var containerInfo = Fields["aContainer"].inspect();
// click the second OK button inside the container
containerInfo.find('OK', { skip: 1 }).click();

// A simpler way of doing the same
Fields["aContainer"].find('OK', { skip: 1 }).click();

// A multi-step approach - only one inspect command is issued before
    the click command
Fields["aContainer"].find(/Button.*Area2/i, { property: 'name' }).find(
    "Cancel").click();

// Find enabled buttons and click them
var visibleButtons = new Field("**/Container").findAll(function(o) {
    return o.enabled == "true" && o.type == "button"; });
for (var i = 0; i < visibleButtons.length; i++) {
  visibleButtons[i].click();
}

// Custom recursive traversal to click the first enabled button
var structure = Fields["aContainer"].inspect();
function search(node) {
  if (node.enabled == "true" && node.type == "button") return node;
  for(var i = 0; i < node.children.length; i++) {
    var hit = search(node.children[i]); // recursive call to dive
        deeper into the structure
    if (hit) return hit;
  }
  return null;
}
search(structure).click();
```

### Input

Input a text value into a textfield/textbox/etc.

#### Parameters

- `text` the text to input
- `options` an optional options object.

  - `useCachedUI` an optional boolean indicating if UI component lookup should use the UI itself or the underlying model. Defaults to **`false`** (underlying model traversal).
  - `file` an optional boolean indicating if the field is an html file input, which requires special treatment. If this is set to true, then the value must be a valid path that points to a file or an exception may be thrown. Only applicable to web apps.
  - `useFieldCache` boolean indicating whether to use a cache to lookup the field - may be significantly faster in some cases - default **`false`**
  - `fieldCacheExpiry` int indicating the useable age in seconds of the field retrieved from the cache (only relevant if `useFieldCache` is **`true`**) - default is configurable as a global option

#### Example

```
Fields["mytextfield"].input("some text");

Fields["myFileField"].input("C:\\some\\file.txt", { file: true });
```

### Native input

Inputs text into a field using native events, i.e. simulating keyboard input. This is useful for fields which does validation (e.g. date-fields or similar). Use only if the `input` method does not work.

#### Parameters

- `text` the text to input - you can use <`backspace`> to indicate a backspace/delete action, as well as <`enter`> and <`tab`>.

#### Example

```
Fields["mydatefield"].inputNative("11112011");
Fields["mydatefield"].inputNative("123<backspace>"); // field will
    contain '12'
```

### Native input with delay

Inputs text into a field using native events with a given delay between each keystroke simulating keyboard input. This is useful for fields which does validation (e.g. date-fields or similar). Use only if the `input` method does not work.

#### Parameters

- `text` the text to input
- `delay` the number of *milliseconds* to wait between each "keystroke"

#### Example

```
1  Fields["mydatefield"].inputNativeWithDelay("some text", 100);
```

### Select

Select a value. This only works for dropdowns, listboxes, tabs and tree-views.

**Note** that for tree-views the value given to this function may be an expression which matches the path to a leaf. E.g. for the following tree:

```
1  tree├──
2   a│
3      └── b│
4      └──      c├──
5   d└──
6   x└──
7         y
```

The node `c` may be selected by:

```
1  Fields["tree"].select("a/b/c");
```

#### Parameters

- `value` the value to select. By default `value` is treated as a regular expression, where characters like `.`, `*` and `(` have special meaning. If you want a literal match you need to surround `value` with << and >>, e.g. `select('<<'+v+'>>')` where `v` is the literal value find in the select options - it will match if the select option contains the given value. To do an exact match use <<< and >>> instead (for *exact* and *contains* matches you can also use the options listed below).
- `options` an optional options object with details regarding the selection.
  - `deadline` the time in ms to wait for the select to fail/succeed. If the select takes longer than the deadline to fail or succeed it will be reported as succeeding to the caller.

  - `reflectionDepth` an option indicating how far the select matching should dive into java objects (eg treeview nodes). Setting this too high may negatively affect performance. Defaults to 0. Use the `inspect` method to determine how to match against this information and what an appropriate (minimal) reflection depth is.
  - `useCachedUI` boolean indicating if UI component lookup should use the UI itself or the underlying model. Defaults to **`false`** (underlying model traversal).
  - `useFieldCache` boolean indicating whether to use a cache to lookup the field - may be significantly faster in some cases - default **`false`**
  - `fieldCacheExpiry` int indicating the useable age in seconds of the field retrieved from the cache (only relevant if `useFieldCache` is **`true`**) - default is configurable as a global option
  - `exact` a bool indicating whether the `value` should match *exactly* (i.e. treated as a literal value and matches with an equals method)
  - `contains` a bool indicating whether the `value` should match by checking if its a *substring*
  - `expand` boolean indicating if the targeted collapsed tree node should be expanded after selection. Only supported in java apps.

#### Example

```
// Select "option1" and use reflectionDepth to to try and find "option1"
Fields["mytree"].select("option1", { reflectionDepth: 2 });

// exact match
Fields["mytree"].select("option1", { exact: true });

// check a checkbox (long form of .check())
Fields["myCheckBox"].select(true);
```

### Select with index

Select a value based in an index. This only works for dropdowns, tabs, listboxes and tree-views.

#### Parameters

- `index` is the index in the combo, listbox or tree to select.
- `options` an optional options object with details regarding the selection.
  - `useCachedUI` boolean indicating if UI component lookup should use the UI itself or the underlying model. Defaults to **`false`** (underlying model traversal).

  - `useFieldCache` boolean indicating whether to use a cache to lookup the field - may be significantly faster in some cases - default **`false`**
  - `fieldCacheExpiry` int indicating the useable age in seconds of the field retrieved from the cache (only relevant if `useFieldCache` is **`true`**) - default is configurable as a global option

#### Example

```
1 Fields["mycombo"].selectIndex(5);
```

### Select with offset

Select a value (with an offset). This only works for dropdowns, tabs, listboxes and tree-views.

#### Parameters

- `value` the value to base selection on. The value needs only to partially match the shown option to be selected, e.g. using "utte" in a list containing the item "butter" will select it.
- `offset` (int) the offset which will be used to do actual selection. E.g. if "1" then the next element (which was found using value will be selected).

#### Example

```
1 Fields["mytree"].selectWithOffset("option1", 1);
```

### Select with offset and skip

Select a value (with an offset and skip). This only works for dropdowns, tabs, listboxes and tree-views.

#### Parameters

- `value` the value to base selection on. The value needs only to partially match the shown option to be selected, e.g. using "utte" in a list containing the item "butter" will select it.
- `offset` (int) the offset which will be used to do actual selection. E.g. if "1" then the next element (which was found using value will be selected).
- `skip` will select the N'th match to start from. E.g. `1` will skip the first match and select the 2nd.

#### Example

```
1  Fields["mytree"].selectWithOffsetAndSkip("option1", 1, 1);
```

If used on e.g. a combobox with options; [“option1”, “option2”, “option1”, “option3”] the code-fragment above will select “option3”. This is done by first looking for all “option1”s. Then skip 1 this will get you the 2nd “option1”, then offset by 1 which will get you “option3”.

### Toggle a checkbox/radiobutton

Check or uncheck a checkbox or radiobutton. Note that radiobuttons don’t always support explicit unchecking. This often requires selecting another radiobutton in the same logical group. A method also exists for getting the checked state of such controls.

#### Check/Uncheck

Fails silently, if the target isn’t a radiobutton or checkbox #### Example

```
1
2  Fields["myCheckbox"].check();
3  Fields["myOtherCheckbox"].uncheck();
```

#### isChecked

This method may throw an error if the field isn’t something that can be checked

```
1
2  var isSelected = Fields["myRadiobutton"].isChecked();
```

### Edit cell

Can be used in a table to edit a given cell.

#### Parameters

- `row` the row in which to find the cell (match any cell in the row) or use an `integer` to denote the row index to edit
- `column` the column in which to find the cell (must match a single column) or use an `integer` to denote the column index to edit

- `value` the value to put into the cell (works with textfield and dropdowns)
- `options` is an optional argument, which can contain:
  - `reflectionDepth` used to finding the value if there is e.g. a combobox in the cell to edit (also see Reflection depth)
  - `useCachedUI` boolean indicating if UI component lookup should use the UI itself or the underlying model. Defaults to **`false`** (underlying model traversal).
  - `useFieldCache` boolean indicating whether to use a cache to lookup the field - may be significantly faster in some cases - default **`false`**
  - `fieldCacheExpiry` int indicating the useable age in seconds of the field retrieved from the cache (only relevant if `useFieldCache` is **`true`**) - default is configurable as a global option

#### Example

Given the following table:

| header 1 | header 2 |
|---|---|
| cell 1 | cell 2 |
| cell 3 | cell 4 |

This command:

```
1 Fields["mytable"].editcell("cell 3", "header 2", "boom");
```

Will result in this table:

| header 1 | header 2 |
|---|---|
| cell 1 | cell 2 |
| cell 3 | *boom* |

The same thing can be accomplished if the indices are known:

```
1 Fields["mytable"].editcell(1, 1, "boom");
```

The method as the `editTable` and `editCell` aliases.

### Highlight

Highlight the given field with the default color.

#### Example

```
1 Fields["myfield"].highlight();
```

### Highlight with color

Highlight the given field with the given color. Available colors are `red`, `green` and `blue`.

#### Parameters

- `color` the highlighting color - `red`, `green` or `blue`.

#### Example

```
1 Fields["myfield"].highlightWithColor("blue");
```

### Lowlight

Cancel a highlight on a field.

#### Example

```
1 Fields["myfield"].lowlight();
```

### Eval

Evaluate javascript in the host app with access to the resolved field UI element in the `element` variable. For more information about evaluating javascript in the host app, see App.eval.

**Example:**

```
// Trigger event on input element. Useful for some web apps
Fields["input"].eval("element.dispatchEvent(new InputEvent('input'));")
    ;

// Call method on parent component in java app
var parentStatus = Fields["myfield"].eval("element.inner.getParent().
    getStatus();");

// Get info about the layout of the java class of the object pointed to
     by the field
var classInfoJson = Fields["myfield"].eval("Class.info(element.inner);"
    );

// Inspect contents of the java object pointed to by the field
var fieldContentJson = Fields["myfield"].eval("Class.inspect(element.
    inner);");

// Retrieve json encoded list of a java JComboBox' options (fictional
    property names)
var optionsJson = Fields["myCombo"].eval("Class.inspect(element, ['
    inner', 'model', 'allOptions], 2);");

// Retrieve deep value from java field - conceptually equivalent to '
    fieldObject.address.streetName'
var streetName = Fields["myfield"].eval("Class.deref(element, ['inner',
     'address', 'streetName']);");
```

For java apps in particular, this version of `eval` provides a shortcut into the app's hierarchy of object instances that might otherwise be hard to reach with `App.eval(..)`.