Ext JS Sencha Docs

Ext.ComponentQuery

Provides searching of Components within Ext.ComponentManager (globally) or a specific Ext.container.Container on the document with a similar syntax to a CSS selector. Returns Array of matching Components, or empty Array.

HIERARCHY

Ext.Base
  Ext.ComponentQuery

REQUIRES

Ext.ComponentManager
Ext.dom.Query

FILES

ComponentQuery.js

Basic Component lookup

Components can be retrieved by using their xtype:

- `component`
- `gridpanel`

Matching by `xtype` matches inherited types, so in the following code, the previous field *of any type which inherits from `TextField`* will be found:

```
prevField = myField.previousNode('textfield');
```

To match only the exact type, pass the "shallow" flag by adding `(true)` to xtype (See AbstractComponent's isXType method):

```
prevTextField = myField.previousNode('textfield(true)');
```

You can search Components by their `id` or `itemId` property, prefixed with a #:

```
#myContainer
```

Component `xtype` and `id` or `itemId` can be used together to avoid possible id collisions between Components of different types:

```
panel#myPanel
```

Traversing Component tree

Components can be found by their relation to other Components. There are several relationship operators, mostly taken from CSS selectors:

- **E F** All descendant Components of E that match F
- **E > F** All direct children Components of E that match F
- **E ^ F** All parent Components of E that match F

Expressions between relationship operators are matched left to right, i.e. leftmost selector is applied first, then if one or more matches are found, relationship operator itself is applied, then next selector expression, etc. It is possible to combine relationship operators in complex selectors:

```
window[title="Input form"] textfield[name=login] ^ form > button[action=submit]
```

That selector can be read this way: Find a window with title "Input form", in that window find a TextField with name "login" at any depth (including subpanels and/or FieldSets), then find an `Ext.form.Panel` that is a parent of the TextField, and in that form find a direct child that is a button with custom property `action` set to value "submit".

Whitespace on both sides of ^ and > operators is non-significant, i.e. can be omitted, but usually is used for clarity.

Searching by Component attributes

Components can be searched by their object property values (attributes). To do that, use attribute matching expression in square brackets:

- `component[autoScroll]` - matches any Component that has `autoScroll` property with any truthy (non-empty, not `false`) value.
- `panel[title="Test"]` - matches any Component that has `title` property set to "Test". Note that if the value does not contain spaces, the quotes are optional.

Attributes can use any of the operators in DomQuery's operators to compare values.

Prefixing the attribute name with an at sign `@` means that the property must be the object's `ownProperty`, not a property from the prototype chain.

Specifications like `[propName]` check that the property is a truthy value. To check that the object has an `ownProperty` of a certain name, regardless of the value use the form `[?propName]`.

The specified value is coerced to match the type of the property found in the candidate Component using Ext.coerce.

If you need to find Components by their `itemId` property, use `#id` form; it will do the same but is easier to read.

Attribute matching operators

The '=' operator will return the results that **exactly** match the specified object property (attribute):

```
Ext.ComponentQuery.query('panel[cls=my-cls]');
```

Will match the following Component:

```
Ext.create('Ext.window.Window', {
    cls: 'my-cls'
});
```

But will not match the following Component, because 'my-cls' is one value among others:

```
Ext.create('Ext.panel.Panel', {
    cls: 'foo-cls my-cls bar-cls'
});
```

You can use the '~=' operator instead, it will return Components with the property that **exactly** matches one of the whitespace-separated values. This is also true for properties that only have *one* value:

```
Ext.ComponentQuery.query('panel[cls~=my-cls]');
```

Will match both Components:

```
Ext.create('Ext.panel.Panel', {
    cls: 'foo-cls my-cls bar-cls'
});

Ext.create('Ext.window.Window', {
    cls: 'my-cls'
});
```

Generally, '=' operator is more suited for object properties other than CSS classes, while '~=' operator will work best with properties that hold lists of whitespace-separated CSS classes.

The '^=' operator will return Components with specified attribute that start with the passed value:

```
Ext.ComponentQuery.query('panel[title^=Sales]');
```

Will match the following Component:

```
Ext.create('Ext.panel.Panel', {
    title: 'Sales estimate for Q4'
});
```

The '$=' operator will return Components with specified properties that end with the passed value:

```
Ext.ComponentQuery.query('field[fieldLabel$=name]');
```

Will match the following Component:

```
Ext.create('Ext.form.field.Text', {
    fieldLabel: 'Enter your name'
});
```

The following test will find panels with their `ownProperty` collapsed being equal to `false`. It will **not** match a collapsed property from the prototype chain.

```
Ext.ComponentQuery.query('panel[@collapsed=false]');
```

Member expressions from candidate Components may be tested. If the expression returns a *truthy* value, the candidate Component will be included in the query:

```
var disabledFields = myFormPanel.query("{isDisabled()}");
```

Such expressions are executed in Component's context, and the above expression is similar to running this snippet for every Component in your application:

```
if (component.isDisabled()) {
    matches.push(component);
}
```

It is important to use only methods that are available in **every** Component instance to avoid run time exceptions. If you need to match your Components with a custom condition formula, you can augment `Ext.Component` to provide custom matcher that will return `false` by default, and override it in your custom classes:

```
Ext.define('My.Component', {
    override: 'Ext.Component',
    myMatcher: function() { return false; }
});

Ext.define('My.Panel', {
    extend: 'Ext.panel.Panel',
    requires: ['My.Component'],     // Ensure that Component override is applied
    myMatcher: function(selector) {
        return selector === 'myPanel';
    }
});
```

After that you can use a selector with your custom matcher to find all instances of `My.Panel`:

```
Ext.ComponentQuery.query("{myMatcher('myPanel')}");
```

However if you really need to use a custom matcher, you may find it easier to implement a custom Pseudo class instead (see below).

Conditional matching

Attribute matchers can be combined to select only Components that match **all** conditions (logical AND operator):

```
Ext.ComponentQuery.query('panel[cls~=my-cls][floating=true][title$="sales data"]');
```

E.g., the query above will match only a Panel-descended Component that has 'my-cls' CSS class *and* is floating *and* with a title that ends with "sales data".

Expressions separated with commas will match any Component that satisfies *either* expression (logical OR operator):

```
Ext.ComponentQuery.query('field[fieldLabel^=User], field[fieldLabel*=password]');
```

E.g., the query above will match any field with field label starting with "User", *or* any field that has "password" in its label.

Pseudo classes

Pseudo classes may be used to filter results in the same way as in Ext.dom.Query. There are five default pseudo classes:

- `not` Negates a selector.

- `first` Filters out all except the first matching item for a selector.
- `last` Filters out all except the last matching item for a selector.
- `focusable` Filters out all except Components which are currently able to recieve focus.
- `nth-child` Filters Components by ordinal position in the selection.

These pseudo classes can be used with other matchers or without them:

```
// Select first direct child button in any panel
Ext.ComponentQuery.query('panel > button:first');

// Select last field in Profile form
Ext.ComponentQuery.query('form[title=Profile] field:last');

// Find first focusable Component in a panel and focus it
panel.down(':focusable').focus();

// Select any field that is not hidden in a form
form.query('field:not(hiddenfield)');
```

Pseudo class `nth-child` can be used to find any child Component by its position relative to its siblings. This class' handler takes one argument that specifies the selection formula as `Xn` or `Xn+Y`:

```
// Find every odd field in a form
form.query('field:nth-child(2n+1)'); // or use shortcut: :nth-child(odd)

// Find every even field in a form
form.query('field:nth-child(2n)');   // or use shortcut: :nth-child(even)

// Find every 3rd field in a form
form.query('field:nth-child(3n)');
```

Pseudo classes can be combined to further filter the results, e.g., in the form example above we can modify the query to exclude hidden fields:

```
// Find every 3rd non-hidden field in a form
form.query('field:not(hiddenfield):nth-child(3n)');
```

Note that when combining pseudo classes, whitespace is significant, i.e. there should be no spaces between pseudo classes. This is a common mistake; if you accidentally type a space between `field` and `:not`, the query will not return any result because it will mean "find *field's children Components* that are not hidden fields...".

Custom pseudo classes

It is possible to define your own custom pseudo classes. In fact, a pseudo class is just a property in `Ext.ComponentQuery.pseudos` object that defines pseudo class name (property name) and pseudo class handler (property value):

```
// Function receives array and returns a filtered array.
Ext.ComponentQuery.pseudos.invalid = function(items) {
    var i = 0, l = items.length, c, result = [];
    for (; i < l; i++) {
        if (!(c = items[i]).isValid()) {
            result.push(c);
        }
    }
    return result;
};

var invalidFields = myFormPanel.query('field:invalid');
if (invalidFields.length) {
    invalidFields[0].getEl().scrollIntoView(myFormPanel.body);
    for (var i = 0, l = invalidFields.length; i < l; i++) {
        invalidFields[i].getEl().frame("red");
    }
}
```

Pseudo class handlers can be even more flexible, with a selector argument used to define the logic:

```
// Handler receives array of itmes and selector in parentheses
Ext.ComponentQuery.pseudos.titleRegex = function(components, selector) {
    var i = 0, l = components.length, c, result = [], regex = new RegExp(selector);
    for (; i < l; i++) {
        c = components[i];
        if (c.title && regex.test(c.title)) {
            result.push(c);
        }
    }
    return result;
}

var salesTabs = tabPanel.query('panel:titleRegex("sales\s+for\s+201[123]")');
```

Be careful when using custom pseudo classes with MVC Controllers: when you use a pseudo class in Controller's `control` or `listen` component selectors, the pseudo class' handler function will be called very often and may slow down your application significantly. A good rule of thumb is to always specify Component xtype with the pseudo class so that the handlers are only called on Components that you need, and try to make the condition checks as cheap in terms of execution time as possible. Note how in the example above, handler function checks that Component *has* a title first, before running regex test on it.

Query examples

Queries return an array of Components. Here are some example queries:

```
// retrieve all Ext.Panels in the document by xtype
var panelsArray = Ext.ComponentQuery.query('panel');

// retrieve all Ext.Panels within the container with an id myCt
```

```
var panelsWithinmyCt = Ext.ComponentQuery.query('#myCt panel');

// retrieve all direct children which are Ext.Panels within myCt
var directChildPanel = Ext.ComponentQuery.query('#myCt > panel');

// retrieve all grids or trees
var gridsAndTrees = Ext.ComponentQuery.query('gridpanel, treepanel');

// Focus first Component
myFormPanel.child(':focusable').focus();

// Retrieve every odd text field in a form
myFormPanel.query('textfield:nth-child(odd)');

// Retrieve every even field in a form, excluding hidden fields
myFormPanel.query('field:not(hiddenfield):nth-child(even)');
```

For easy access to queries based from a particular Container see the Ext.container.Container.query, Ext.container.Container.down and Ext.container.Container.child methods. Also see Ext.Component.up.

Properties

Instance properties	Defined By
$className : String PRIVATE Defaults to: 'Ext.Base'	Ext.Base
configMap : Object PRIVATE Defaults to: {}	Ext.Base
initConfigList : Array PRIVATE Defaults to: []	Ext.Base
initConfigMap : Object PRIVATE Defaults to: {}	Ext.Base
isInstance : Boolean PRIVATE Defaults to: true	Ext.Base
self : Ext.Class PROTECTED Get the reference to the current class from which this object was instantiated. Unlike statics, this.self is scope-dependent and it's meant to be used for dynamic inheritance. See statics for a detailed comparison	Ext.Base

```
Ext.define('My.Cat', {
    statics: {
        speciesName: 'Cat' // My.Cat.speciesName = 'Cat'
    },

    constructor: function() {
        alert(this.self.speciesName); // dependent on 'this'
    },

    clone: function() {
        return new this.self();
    }
});


Ext.define('My.SnowLeopard', {
    extend: 'My.Cat',
    statics: {
        speciesName: 'Snow Leopard'         // My.SnowLeopard.speciesName = 'Snow Leopard'
    }
});

var cat = new My.Cat();                     // alerts 'Cat'
var snowLeopard = new My.SnowLeopard();     // alerts 'Snow Leopard'

var clone = snowLeopard.clone();
alert(Ext.getClassName(clone));             // alerts 'My.SnowLeopard'
```

Static properties	Defined By
$onExtended : Array PRIVATE STATIC Defaults to: []	Ext.Base

Methods

Instance methods	Defined By
callOverridden(args) : Object DEPRECATED PROTECTED Call the original method that was previously overridden with override	Ext.Base

```
Ext.define('My.Cat', {
    constructor: function() {
        alert("I'm a cat!");
    }
});

My.Cat.override({
    constructor: function() {
        alert("I'm going to be a cat!");
```

```
        this.callOverridden();

        alert("Meeeeoooowwww");
    }
});

var kitty = new My.Cat(); // alerts "I'm going to be a cat!"
                          // alerts "I'm a cat!"
                          // alerts "Meeeeoooowwww"
```

This method has been **DEPRECTED**

as of 4.1. Use callParent instead.

Parameters

- `args` : Array/Arguments

  The arguments, either an array or the `arguments` object from the current method, for example: `this.callOverridden(arguments)`

Returns

- `Object`

  Returns the result of calling the overridden method

callParent(`args`) : Object **PROTECTED** Ext.Base

Call the "parent" method of the current method. That is the method previously overridden by derivation or by an override (see Ext.define).

```
Ext.define('My.Base', {
    constructor: function (x) {
        this.x = x;
    },

    statics: {
        method: function (x) {
            return x;
        }
    }
});

Ext.define('My.Derived', {
    extend: 'My.Base',

    constructor: function () {
        this.callParent([21]);
    }
});

var obj = new My.Derived();

alert(obj.x);  // alerts 21
```

This can be used with an override as follows:

```
Ext.define('My.DerivedOverride', {
    override: 'My.Derived',

    constructor: function (x) {
        this.callParent([x*2]); // calls original My.Derived constructor
    }
});

var obj = new My.Derived();

alert(obj.x);  // now alerts 42
```

This also works with static methods.

```
Ext.define('My.Derived2', {
    extend: 'My.Base',

    statics: {
        method: function (x) {
            return this.callParent([x*2]); // calls My.Base.method
        }
    }
});

alert(My.Base.method(10);     // alerts 10
alert(My.Derived2.method(10); // alerts 20
```

Lastly, it also works with overridden static methods.

```
Ext.define('My.Derived2Override', {
    override: 'My.Derived2',

    statics: {
        method: function (x) {
            return this.callParent([x*2]); // calls My.Derived2.method
        }
    }
});

alert(My.Derived2.method(10); // now alerts 40
```

To override a method and replace it and also call the superclass method, use callSuper. This is often done to patch a method to fix a bug.

Parameters

- `args` : Array/Arguments

  The arguments, either an array or the `arguments` object from the current method, for example: `this.callParent(arguments)`

Returns

- `Object`

  Returns the result of calling the parent method

callSuper(`args`) : Object PROTECTED — Ext.Base

This method is used by an override to call the superclass method but bypass any overridden method. This is often done to "patch" a method that contains a bug but for whatever reason cannot be fixed directly.

Consider:

```
Ext.define('Ext.some.Class', {
    method: function () {
        console.log('Good');
    }
});

Ext.define('Ext.some.DerivedClass', {
    method: function () {
        console.log('Bad');

        // ... logic but with a bug ...

        this.callParent();
    }
});
```

To patch the bug in `DerivedClass.method`, the typical solution is to create an override:

```
Ext.define('App.paches.DerivedClass', {
    override: 'Ext.some.DerivedClass',

    method: function () {
        console.log('Fixed');

        // ... logic but with bug fixed ...

        this.callSuper();
    }
});
```

The patch method cannot use `callParent` to call the superclass `method` since that would call the overridden method containing the bug. In other words, the above patch would only produce "Fixed" then "Good" in the console log, whereas, using `callParent` would produce "Fixed" then "Bad" then "Good".

Parameters

- `args` : Array/Arguments

  The arguments, either an array or the `arguments` object from the current method, for example: `this.callSuper(arguments)`

Returns

- `Object`

  Returns the result of calling the superclass method

configClass( ) PRIVATE — Ext.Base

destroy( ) PRIVATE — Ext.Base

Overrides: Ext.state.Stateful.destroy, Ext.AbstractComponent.destroy, Ext.AbstractPlugin.destroy

getConfig(`name`) PRIVATE — Ext.Base

Parameters

- `name` : Object

getInitialConfig(`[name]`) : Object/Mixed — Ext.Base

Returns the initial configuration passed to constructor when instantiating this class.

Parameters

- `name` : String (optional)

  Name of the config option to return.

Returns

- `Object/Mixed`

  The full config object or a single config value when `name` parameter specified.

hasConfig(`config`) PRIVATE — Ext.Base

Parameters

- `config` : Object

initConfig(`config`) : Ext.Base CHAINABLE PROTECTED — Ext.Base

Initialize configuration for this class. a typical example:

```
Ext.define('My.awesome.Class', {
```

```
    // The default config
    config: {
        name: 'Awesome',
        isAwesome: true
    },

    constructor: function(config) {
        this.initConfig(config);
    }
});

var awesome = new My.awesome.Class({
    name: 'Super Awesome'
});

alert(awesome.getName()); // 'Super Awesome'
```

Parameters

- `config` : Object

Returns

- `Ext.Base`

  this

---

is(`component, selector`) : Boolean — Ext.ComponentQuery

Tests whether the passed Component matches the selector string.

Parameters

- `component` : Ext.Component

  The Component to test

- `selector` : String

  The selector string to test against.

Returns

- `Boolean`

  True if the Component matches the selector.

---

onConfigUpdate(`names, callback, scope`) PRIVATE — Ext.Base

Parameters

- `names` : Object
- `callback` : Object
- `scope` : Object

---

query(`selector, [root]`) : Ext.Component[] — Ext.ComponentQuery

Returns an array of matched Components from within the passed root object.

This method filters returned Components in a similar way to how CSS selector based DOM queries work using a textual selector string.

See class summary for details.

Parameters

- `selector` : String

  The selector string to filter returned Components

- `root` : Ext.container.Container (optional)

  The Container within which to perform the query. If omitted, all Components within the document are included in the search.

  This parameter may also be an array of Components to filter according to the selector.

Returns

- `Ext.Component[]`

  The matched Components.

---

setConfig(`config, applyIfNotSet`) : Ext.Base CHAINABLE PRIVATE — Ext.Base

Parameters

- `config` : Object
- `applyIfNotSet` : Object

Returns

- `Ext.Base`

  this

---

statics( ) : Ext.Class PROTECTED — Ext.Base

Get the reference to the class from which this object was instantiated. Note that unlike self, `this.statics()` is scope-independent and it always returns the class from which it was called, regardless of what `this` points to during run-time

```
Ext.define('My.Cat', {
    statics: {
        totalCreated: 0,
        speciesName: 'Cat' // My.Cat.speciesName = 'Cat'
    },
```

```
    constructor: function() {
        var statics = this.statics();

        alert(statics.speciesName);     // always equals to 'Cat' no matter what 'this' refers to
                                        // equivalent to: My.Cat.speciesName

        alert(this.self.speciesName);   // dependent on 'this'

        statics.totalCreated++;
    },

    clone: function() {
        var cloned = new this.self;                      // dependent on 'this'

        cloned.groupName = this.statics().speciesName;   // equivalent to: My.Cat.speciesName

        return cloned;
    }
});


Ext.define('My.SnowLeopard', {
    extend: 'My.Cat',

    statics: {
        speciesName: 'Snow Leopard'     // My.SnowLeopard.speciesName = 'Snow Leopard'
    },

    constructor: function() {
        this.callParent();
    }
});

var cat = new My.Cat();                 // alerts 'Cat', then alerts 'Cat'

var snowLeopard = new My.SnowLeopard(); // alerts 'Cat', then alerts 'Snow Leopard'

var clone = snowLeopard.clone();
alert(Ext.getClassName(clone));         // alerts 'My.SnowLeopard'
alert(clone.groupName);                 // alerts 'Cat'

alert(My.Cat.totalCreated);             // alerts 3
```

Returns

- `Ext.Class`

Static methods	Defined By

addConfig(`config`) PRIVATE STATIC — Ext.Base

Parameters

- `config` : Object

addInheritableStatics(`members`) CHAINABLE PRIVATE STATIC — Ext.Base

Parameters

- `members` : Object

addMember(`name, member`) CHAINABLE PRIVATE STATIC — Ext.Base

Parameters

- `name` : Object
- `member` : Object

addMembers(`members`) CHAINABLE STATIC — Ext.Base

Add methods / properties to the prototype of this class.

```
Ext.define('My.awesome.Cat', {
    constructor: function() {
        ...
    }
});

 My.awesome.Cat.addMembers({
     meow: function() {
        alert('Meowww...');
     }
 });

 var kitty = new My.awesome.Cat;
 kitty.meow();
```

Parameters

- `members` : Object

addStatics(`members`) : Ext.Base CHAINABLE STATIC — Ext.Base

Add / override static properties of this class.

```
Ext.define('My.cool.Class', {
    ...
```

```
});

My.cool.Class.addStatics({
    someProperty: 'someValue',      // My.cool.Class.someProperty = 'someValue'
    method1: function() { ... },    // My.cool.Class.method1 = function() { ... };
    method2: function() { ... }     // My.cool.Class.method2 = function() { ... };
});
```

Parameters

- `members` : Object

Returns

- `Ext.Base`

  this

---

addXtype(`xtype`) CHAINABLE PRIVATE STATIC Ext.Base

Parameters

- `xtype` : Object

---

borrow(`fromClass, members`) : Ext.Base CHAINABLE PRIVATE STATIC Ext.Base

Borrow another class' members to the prototype of this class.

```
Ext.define('Bank', {
    money: '$$$',
    printMoney: function() {
        alert('$$$$$$$');
    }
});

Ext.define('Thief', {
    ...
});

Thief.borrow(Bank, ['money', 'printMoney']);

var steve = new Thief();

alert(steve.money); // alerts '$$$'
steve.printMoney(); // alerts '$$$$$$$'
```

Parameters

- `fromClass` : Ext.Base

  The class to borrow members from

- `members` : Array/String

  The names of the members to borrow

Returns

- `Ext.Base`

  this

---

create( ) : Object STATIC Ext.Base

Create a new instance of this Class.

```
Ext.define('My.cool.Class', {
    ...
});

My.cool.Class.create({
    someConfig: true
});
```

All parameters are passed to the constructor of the class.

Returns

- `Object`

  the created instance.

Overrides: Ext.layout.Layout.create

---

createAlias(`alias, origin`) STATIC Ext.Base

Create aliases for existing prototype methods. Example:

```
Ext.define('My.cool.Class', {
    method1: function() { ... },
    method2: function() { ... }
});

var test = new My.cool.Class();

My.cool.Class.createAlias({
    method3: 'method1',
    method4: 'method2'
});

test.method3(); // test.method1()

My.cool.Class.createAlias('method5', 'method3');
```

```
test.method5(); // test.method3() -> test.method1()
```

Parameters

- `alias` : String/Object
  The new method name, or an object to set multiple aliases. See flexSetter
- `origin` : String/Object
  The original method name

---

extend(`config`) PRIVATE STATIC — Ext.Base

Parameters

- `config` : Object

---

getName( ) : String STATIC — Ext.Base

Get the current class' name in string format.

```
Ext.define('My.cool.Class', {
    constructor: function() {
        alert(this.self.getName()); // alerts 'My.cool.Class'
    }
});

My.cool.Class.getName(); // 'My.cool.Class'
```

Returns

- `String`
  className

---

implement( ) DEPRECATED STATIC — Ext.Base

Adds members to class.

This method has been **DEPRECTED** since 4.1

Use addMembers instead.

---

mixin(`name, mixinClass`) PRIVATE STATIC — Ext.Base

Used internally by the mixins pre-processor

Parameters

- `name` : Object
- `mixinClass` : Object

---

onExtended(`fn, scope`) CHAINABLE PRIVATE STATIC — Ext.Base

Parameters

- `fn` : Object
- `scope` : Object

---

override(`members`) : Ext.Base CHAINABLE DEPRECATED STATIC — Ext.Base

Override members of this class. Overridden methods can be invoked via callParent.

```
Ext.define('My.Cat', {
    constructor: function() {
        alert("I'm a cat!");
    }
});

My.Cat.override({
    constructor: function() {
        alert("I'm going to be a cat!");

        this.callParent(arguments);

        alert("Meeeeoooowwww");
    }
});

var kitty = new My.Cat(); // alerts "I'm going to be a cat!"
                          // alerts "I'm a cat!"
                          // alerts "Meeeeoooowwww"
```

As of 4.1, direct use of this method is deprecated. Use Ext.define instead:

```
Ext.define('My.CatOverride', {
    override: 'My.Cat',
    constructor: function() {
        alert("I'm going to be a cat!");

        this.callParent(arguments);

        alert("Meeeeoooowwww");
    }
});
```

The above accomplishes the same result but can be managed by the Ext.Loader which can properly order the override and its target class and the build

process can determine whether the override is needed based on the required state of the target class (My.Cat).

This method has been **DEPRECTED** since 4.1.0

Use Ext.define instead

Parameters

- `members` : Object

  The properties to add to this class. This should be specified as an object literal containing one or more properties.

Returns

- `Ext.Base`

  this class

triggerExtended( ) PRIVATE STATIC Ext.Base