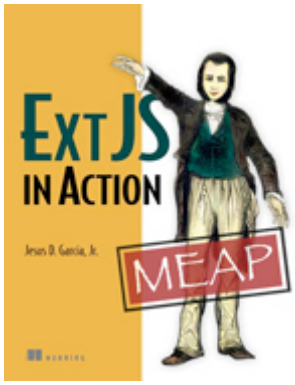
MANNING PUBLICATIONS

The Component model and lifecycle

An article from



Ext JS in Action
EARLY ACCESS EDITION

Jesus Garcia
MEAP Release: February 2009
Softbound print: November 2010 (est.) | 425 pages
ISBN: 9781935182115

This article is taken from the book Ext JS in Action. *The author discusses the fundamental UI building block, the Component class, and how it serves as the central model for all UI widgets by implementing a template for standard behaviors known as the Component lifecycle.*

We'll explore the deep caverns of the fundamental UI building block, the Component class, and learn how it serves as the central model for all UI widgets by implementing a template for standard behaviors known as the Component lifecycle.

The Component model

The Ext Component model is a centralized model that provides many of the essential `Component`-related tasks, which include a set of rules dictating how the `Component` instantiates, renders, and is destroyed, known as the `Component` lifecycle.

All UI widgets are subclasses of `Ext.Component`, which means that all of the widgets conform to the rules dictated by the model. Figure 1 partially depicts how many items subclass `Component`, directly or indirectly.

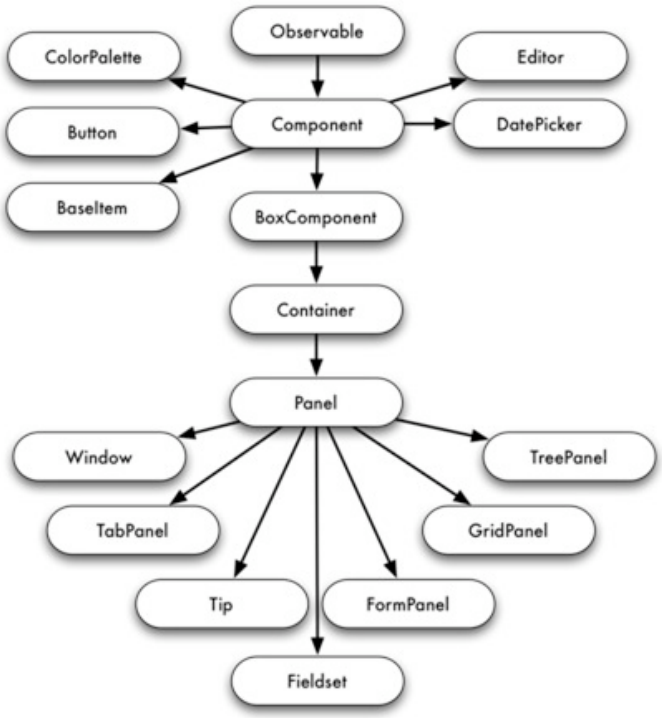


Figure 1 This illustration of the Ext class hierarchy focuses on some of the common subclasses of `Ext.Component` and depicts how widely the Component model is used in the framework.

Knowing how each UI widget is going to behave introduces stability and predictability into the framework, which I enjoy. The Component model also supports direct instantiation of classes or deferred instantiation, which is known as XTypes. Knowing which to use when can enhance the responsiveness of your application.

XTypes and Component Manager

Ext 2.0 introduced a radical new concept known as an `XType`, which allows for lazy instantiation of `Component`s, which can speed up complex user interfaces and can clean up our code quite a bit.

In short, an `XType` is nothing more than a plain JavaScript object, which generally contains an `xtype` property with a `string` value denoting which class the `XType` is for. Here's a quick example of an `XType` in action:

```
var myPanel = {
    xtype    : 'panel',
    height   : 100,
    width    : 100,
    html     : 'Hello!'
};
```

In this configuration object, `myPanel` is an `XType` configuration object that would be used to configure an `Ext.Panel` widget. This works because just about every widget is registered to the `Ext.ComponentMgr` class with a unique string key and a reference to that class, which is then referred to as an `XType`. At the tail end of each Ext UI widget class, you'll find the registration of that widget in the `Ext.ComponentMgr`.

Registration of a `Component` is simple:

```
Ext.reg('myCustomComponent', myApp.customClass);
```

The act of registering a `Component` to the `ComponentMgr` appends or replaces the new `Component` to the internal reference map of the `ComponentMgr` singleton. Once registration is complete, you can specify your custom `Component` as an `XType`:

```
new Ext.Panel({
    ...
    items : {
        xtype : 'myCustomComponent',
```

```
            ...
        }
    });
```

When a visual `Component`—which can contain children—is initialized, it looks to see if it has `this.items` and will inspect `this.items` for `XType` configuration objects. If any are found, it will attempt to create an instance of that `Component` using `ComponentMgr.create`. If the `xtype` property isn't defined in the configuration object, the visual `Component` will use its `defaultType` property when calling `ComponentMgr.create`.

I realize that this may sound a tad confusing at first. I think you can better understand this concept if we exercise it. To do this, we'll create a window with an accordion layout that includes two children, one of which won't contain an `xtype` property. First, let's create our configuration objects for two of the children:

```
var panel1 = {
    xtype : 'panel',
    title : 'Plain Panel',
    html  : 'Panel with an xtype specified'
};

var panel2 = {
    title : 'Plain Panel 2',
    html  : 'Panel with <b>no</b> xtype specified'
};
```

Notice that `panel1` has an explicit `xtype` value of `'panel'`, which, in turn, will be used to create an instance of `Ext.Panel`. Objects `panel1` and `panel2` are similar but have two distinct differences. Object `panel1` has an `xtype` specified, whereas `panel2` doesn't.

Next, we'll create our window, which will use these `xtypes`:

```
new Ext.Window({
    width        : 200,
    height       : 150,
    title        : 'Accordion window',
    layout       : 'accordion',
    border       : false,
    layoutConfig : {
        animate : true
    },
    items : [
        panel1,
        panel2
    ]
}).show();
```

In our new instantiation of `Ext.Window`, we pass items, which are an array of references to the two configuration objects we created earlier. The rendered window should appear as illustrated in figure 2. Clicking a collapsed panel will expand and collapse any other expanded panels, and clicking an expanded panel will collapse it.

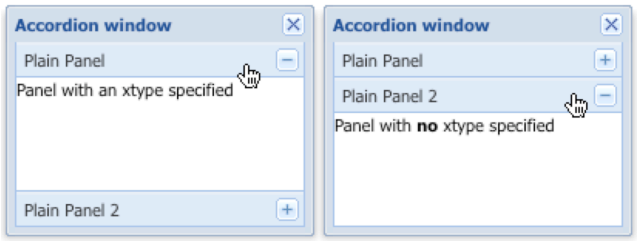


Figure 2 The results of our `XType` exercise: an Ext `Window`, which has two child panels derived from `XType` configuration objects

One of the lesser-known advantages of using `XTypes` is developing somewhat cleaner code. Because you can use plain object notation, you can specify all of your `XType` child items inline, resulting in cleaner and more streamlined code. Here is the previous example reformatted to include all of its children inline:

```
new Ext.Window({
    width        : 200,
    height       : 150,
    title        : 'Accordion window',
```

```
        layout       : 'accordion',
        border       : false,
        layoutConfig : {
            animate : true
        },
        items : [
            {
                xtype : 'panel',
                title : 'Plain Panel',
                html  : 'Panel with an xtype specified'
            },
            {
                title : 'Plain Panel 2',
                html  : 'Panel with <b>no</b> xtype specified'
            }
        ]
    }).show();
```

As you can see, we've included all of the child configuration items in line with the `Window` configuration object. The performance enhancements with using `XType`s can't be seen in such a simple example. The biggest `XType`-based performance gains come from bigger applications, where there are a rather large number of `Component`s to be instantiated.

`Component`s also contain another performance-enhancing feature, lazy rendering. This means a `Component` is rendered only when necessary.

Component rendering

The `Ext.Component` class supports both direct and lazy (on-demand) render models. Direct rendering can happen when a subclass of `Component` is instantiated with either the `renderTo` or `applyTo` attribute, where `renderTo` points to a reference from which the `Component` renders itself, and `applyTo` references an element that has HTML that's structured in such a way that allows the `Component` to create its own child elements based on the referenced HTML. You'd typically use these parameters when you want a `Component` to be rendered upon instantiation, for instance:

```
    var myPanel = new Ext.Panel({
        renderTo : document.body,
        height   : 50,
        width    : 150,
        title    : 'Lazy rendered Panel',
        frame    : true
    });
```

The result of this code would be the immediate render of the `Ext.Panel`, which sometimes is favorable and other times not. The times where it's not favorable can be when you want to *defer* rendering to another time in code execution or the `Component` is a child of another.

If you want to defer the rendering of the `Component`, omit the `renderTo` and `applyTo` attributes and call the `Components render` method when you (or your code) deem it necessary:

```
    var myPanel = new Ext.Panel({
        height : 50,
        width  : 150,
        title  : 'Lazy rendered Panel',
        frame  : true
    });

    // ... some business logic...

    myPanel.render(document.body);
```

In this example, you instantiate an instance of `Ext.Panel` and create a reference to it, `myPanel`. After some hypothetic application logic, you call `myPanel.render` and pass a reference to `document.body`, which renders the panel to the document body.

You could also pass an ID of an element to the `render` method:

```
myPanel.render('someDivId');
```

When passing an element ID to the `render` method, `Component` will use that ID with `Ext.get` to manage that element, which gets stored in its local `el` property. (You can access a widget's `el` property or use its *accessor* method, `getEl`, to get obtain the reference.)

There's one major exception to this rule, however. You *never* specify `applyTo` or `renderTo` when the `Component` is a child of another. `Component`s that *contain* other `Component`s have a parent-child relationship, which is known as the Container model. If a `Component` is a child of another `Component`, it's specified in the `items` attribute of the configuration object, and its parent will manage the call to its `render` method when required. This is known as lazy or deferred rendering.

Now, we'll move on to discussing the `Component` lifecycle, which details how the `Component`s are created, rendered, and eventually destroyed. Learning how each phase works will better prepare you for building robust and dynamic interfaces and can assist in troubleshooting issues.

The Component lifecycle

Ext `Component`s, like everything in the real world, have a lifecycle where they're created, used, and destroyed. This lifecycle is broken up into three major phases: initialization, render, and destruction, as diagrammed in figure 3.

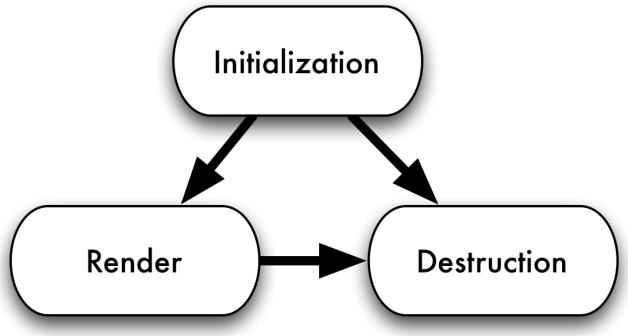


Figure 3 The Ext `Component` lifecycle always starts with initialization and always ends with destruction. The `Component` need not enter the render phase to be destroyed.

To better utilize the framework, you must understand in finer detail how the lifecycle works. This is especially important if you'll be building extensions, plug-ins, or composite `Component`s. Quite a few steps take place at each phase of the lifecycle, which is controlled by the base class, `Ext.Component`.

Initialization

The initialization phase is when a `Component` is born. All of the necessary configuration settings, event registration, and pre-render processes take place in this phase, as illustrated in figure 4.

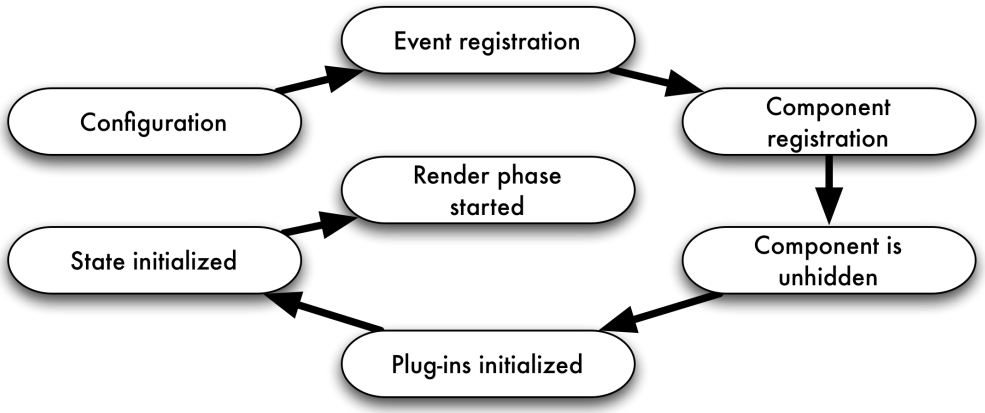


Figure 4 The initialization phase of the `Component` lifecycle executes important steps such as event and `Component` registration as well as the calling of the `initComponent` method. It's important to remember that a `Component` can be instantiated but may not be rendered.

Let's explore each step of the initialization phase:

1. *The configuration is applied*— When instantiating an instance of a `Component`, you pass a configuration object, which contains all of the necessary parameters and references to allow the `Component` to do what it's designed to do. This is done within the first few lines of the `Ext.Component` base class.
2. *Registration of the base* `Component` *events*—Per the Component model, each subclass of `Ext.Component` has, by default, a set of core events that are fired from the base class. These are fired before and after some behaviors occur: enable/disable, show, hide, render, destroy, state restore, and state save. The before events are fired and tested for a successful return of a registered event handler and will cancel the behavior before any real action takes place. For instance, when `myPanel.show` is called, it fires the `beforeshow` event, which will execute any methods registered for that event. If the `beforeshow` event handler returns `false`, `myPanel` doesn't show.
3. `ComponentMgr` *registration*—Each `Component` that's instantiated is registered with the `ComponentMgr` class with a unique Ext-generated string ID. You can choose to override the Ext-generated ID by passing an `id` parameter in the configuration object passed to a constructor. The main caveat is that, if a registration request occurs with a non-unique registration ID, the newest registration will override the previous one. Be careful to use unique IDs if you plan to use your own ID scheme.
4. `initComponent` *is executed*—The `initComponent` method is where a lot of work occurs for `Component` subclasses like the registration of subclass-specific events, references to data stores, and the creation of child `Component`s. `initComponent` is used as a supplement to the constructor and is used as the main point to extending `Component` or any subclass thereof.
5. *Plug-ins are initialized*—If plug-ins are passed in the configuration object to the constructor, their `init` method is called, with the parent `Component` passed as a reference. It's important to remember that the plug-ins are called in the order in which they're referenced.
6. *State is initialized*—If the `Component` is state aware, it will register its state with the global `StateManager` class. Many Ext widgets are state aware.
7. `Component` *is rendered*—If the `renderTo` or `applyTo` parameter is passed into the constructor, the render phase begins at this time; otherwise, the `Component` lies dormant, awaiting its `render` method to be called.

This phase of a `Component`'s life is usually the fastest because all of the work is done in JavaScript. It's particularly important to remember that the `Component` doesn't have to be rendered to be destroyed.

Render

The render phase is the one where you get the visual feedback that a `Component` has been successfully initialized. If the initialization phase fails for whatever reason, the `Component` may not render correctly or at all. For complex `Component`s, this is where a lot of CPU cycles get eaten up, where the browser is required to paint the screen and computations take place to allow all of the items for the `Component` to be properly laid out and sized. Figure 5 illustrates the steps of the render phase.

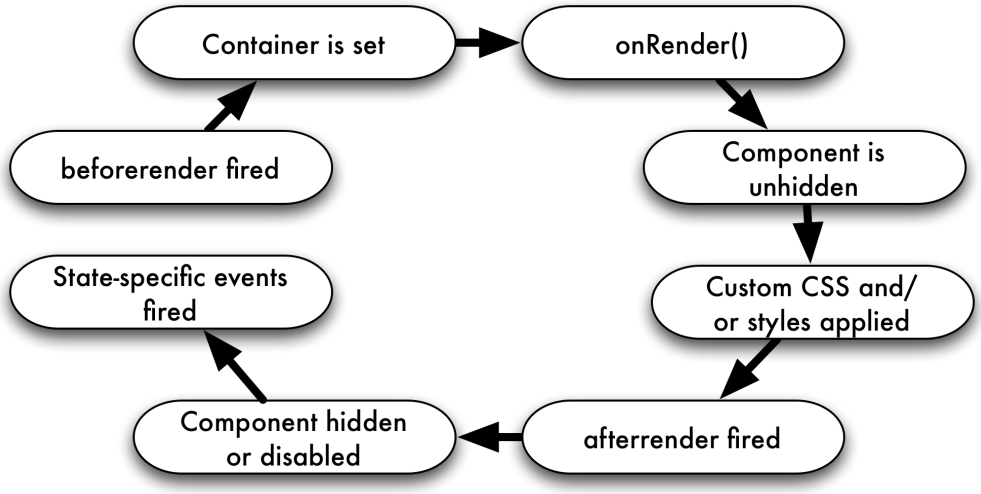


Figure 5 The render phase of the a `Component`'s life can utilize a lot of CPU because it requires elements to be added to the DOM and calculations to be performed to properly size and manage them.

If `renderTo` or `applyTo` isn't specified, a call to the `render` method must be made, which triggers this phase. If the `Component` is not a child of another Ext `Component`, then your code must call the `render` method, passing a reference of the DOM element:

```
someComponent.render('someDivId');
```

If the `Component` is a child of another `Component`, then its `render` method will be called by the parent `Component`. Let's explore the different steps of the render phase:

1. `beforerender` *is fired*—The `Component` fires the `beforerender` event and checks the return of any of the registered event handlers. If a registered event handler returns `false`, the `Component` halts the rendering behavior. Recall that step 2 of the initialization phase registers core events for subclasses of `Component` and that "before" events can halt execution behaviors.
2. *The* `Container` *is set*—A `Component` needs a place to live, and that place is known as its `Container`. If you specify a `renderTo` reference to an element, the `Component` adds a single child `div` element to the referenced element, known as its `Container`, and renders the `Component` inside that newly appended child. If an `applyTo` element is specified, the element referenced in the `applyTo` parameter becomes the `Component`'s `Container`, and the `Component` appends only those items to the referenced element that are required to render it. The DOM element referenced in `applyTo` will then be fully managed by the `Component`. You generally pass neither when the `Component` is a child of another `Component`, in which the container is the parent `Component`. It's important to note that you should pass only `renderTo` or `applyTo` and not both.
3. `onRender` *is executed*—This is a crucial step for `Component` subclasses, where all of the DOM elements are inserted to get the `Component` rendered and painted on the screen. Each subclass is expected to call its `superclass.onRender` first when extending `Ext.Component` or any subclass thereafter, which ensures that the `Ext.Component` base class can insert the core DOM elements needed to render a `Component`.

4. *The* `Component` *is unhidden*—Many `Component`s are rendered hidden using the default Ext CSS class, like `'x-hidden'`. If the `autoShow` property is set, any Ext CSS classes that are used for hiding `Component`s are removed. It's important to note that this step doesn't fire the `show` event, so any listeners for that event won't be fired.
5. *Custom CSS classes or styles are applied*—Custom CSS classes and styles can be specified upon `Component` instantiation by means of the `cls` and `style` parameters. If these parameters are set, they're applied to the `Container` for the `Component`. It's suggested that you use `cls` instead of `style`, because CSS inheritance rules can be applied to the `Component`'s children.
6. *The* `render` *event is fired*—At this point, all necessary elements have been injected into the DOM and styles applied. The `render` event is fired, triggering any registered event handlers for this event.
7. `afterRender` *is executed*—The `afterRender` event is a crucial post-render method that's automatically called by the `render` method within the `Component` base class and can be used to set sizes for the `Container` or perform any other post-render functions. All subclasses of `Component` are expected to call their `superclass.afterRender` method.
8. *The* `Component` *is hidden and/or disabled*—If either `hidden` or `disabled` is specified as `true` in the configuration object, the `hide` or `disable` method is called, which fires its respective `before<action>` event, which is cancelable. If both are `true` and the `before<action>` registered event handlers don't return `false`, the `Component` is both hidden and disabled.
9. *State-specific events are fired*—If the `Component` is state aware, it will initialize its state-specific events with its `Observable` and register the `this.saveEvent` internal method as the handler for each of those state events.
10. Once the render phase is complete, unless the `Component` is disabled or hidden, it's ready for user interaction. It stays alive until its `destroy` method is called, in which it then starts its destruction phase.

The render phase is generally where a `Component` spends most of its life until it meets its demise with the destruction phase.

Destruction

As in real life, the death of a `Component` is a crucial phase in its life. Destruction of a `Component` performs critical tasks, such as removing itself and any children from the DOM tree, deregistration of the `Component` from `ComponentMgr`, and deregistration of event listeners, as depicted in figure 6.

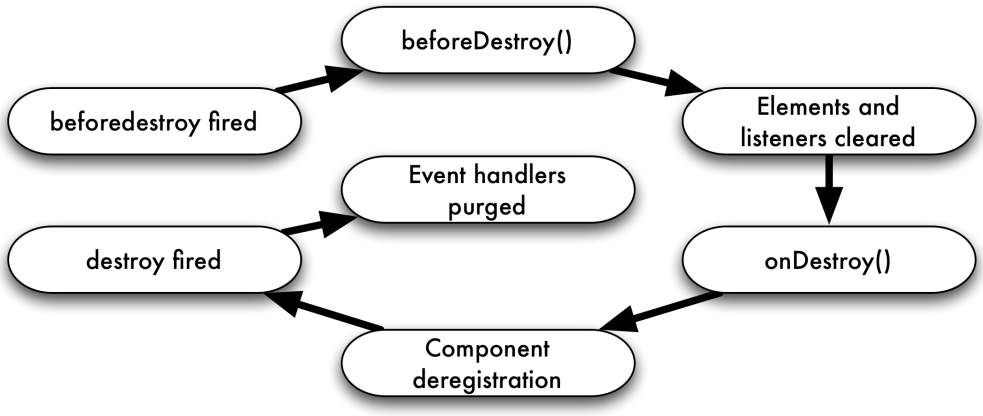


Figure 6 The destruction portion of a `Component`'s life is equally as important as its initialization because event listeners and DOM elements must be deregistered and removed, reducing overall memory usage.

The `Component`'s `destroy` method could be called by a parent `Container` or by your code. Here are the steps in this final phase of a `Component`'s life:

1. `beforedestroy` *is fired*—This, like many `before<action>` events, is a cancelable event, preventing the `Component`'s destruction if its event handler returns `false`.
2. `beforeDestroy` *is called*—This method is first to be called within the `Component`'s `destroy` method and is the perfect opportunity to remove any non-`Component` items, such as toolbars or buttons. A `Component` subclass is expected to call its `superclass.beforeDestroy`.
3. `Element` *and* `Element` *listeners are purged*—If a `Component` has been rendered, any handlers registered to its `Element` are removed and the `Element` is removed from the DOM.
4. `onDestroy` *is called*—Although the `Component` class itself doesn't perform any actions within the `onDestroy` method, subclasses are expected to use this to perform any post-destruction actions, such as removal of data stores. The `Container` class, which subclasses `Component` indirectly, manages the destruction of all registered children by `onDestroy` method, alleviating the developer from this task.
5. `Component` *is unregistered from* `ComponentMgr`—The reference for this `Component` in the `ComponentMgr` class is removed.
6. *The* `destroy` *event is fired*—Any registered event handlers are triggered by this event, which signals that the `Component` is no longer in the DOM.
7. `Component`*'s event handlers are purged*—All event handlers are deregistered from the `Component`.

And there you have it, an in-depth look at the `Component` lifecycle, which is one of the features of the Ext framework that makes it so powerful and successful.

Do not dismiss the destruction portion of a `Component`'s lifecycle if you plan on developing your own custom `Component`s. Many developers have gotten into trouble when they've ignored this crucial step and have code that has left artifacts such as data `Store`s that continuously poll web servers, or event listeners that are expecting an `Element` to be in the DOM were not cleaned properly and cause exceptions and the halt of execution of a crucial branch of logic.

Summary

We took an in-depth look at the Component model, which gives the Ext framework a unified method of managing instances of Components. The Component lifecycle is one of the most important concepts for the UI portion of the framework.