# How to use classes in Sencha Touch 2

Watch the Class System video from SenchaCon 2011

**Contents**

1. Dependencies and Dynamic Loading
2. Naming Conventions
3. Working with classes in Sencha Touch 2.0
4. Error Handling and debugging
5. Further Reading

Sencha Touch 2 uses the state of the art class system developed for Ext JS 4. It makes it easy to create new classes in JavaScript, providing inheritance, dependency loading, mixins, powerful configuration options, and lots more.

Sencha Touch 2는 ExtJS 4에 개발되어진 클래스 시스템을 사용한다. 이 클래스 시스템은 상속, 의존되는 파일 포함, mixins, 구성 옵션 등과 같은 기능을 제공하며, 자바스크립트에서 새로운 클래스를 쉽게 만들 수 있도록 도와준다.

At its simplest, a class is just an object with some functions and properties attached to it. For instance, here's a class for an animal, recording its name and a function that makes it speak:

클래스는 간단히 말해서 기능(function)과 속성(attribute)을 가진 개체이다. 여기 "동물"이란 클래스가 있는데, 이 클래스에서는 자신의 이름을 가질 수 있고, 말을 한다.

```
Ext.define('Animal',{
    config:{
        name:null
    },

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

    speak:function(){
        alert('grunt');
    }});
```

We now have a class called `Animal`, where each animal can have a name and speak. To create a new instance of animal, we just use Ext.create:

지금 “Animal”이란 클래스를 만들었다. 이 클래스는 자신의 이름을 가질 수 있고 말을 할 수 있는 기능을 구현하였다. 그러나 아직 기뻐하기에는 이르다. 이는 단지 껍데기일 뿐이다. 이 껍데기에 생명을 넣어줘야 하는데, 이 때 Ext.create를 사용하면 쉽게 해결된다.

```
var bob =Ext.create('Animal',{
    name:'Bob'});

bob.speak();//alerts 'grunt'
```

Here we created an Animal called Bob and commanded it to speak. Now that we've created a class and instantiated it, we can start improving what we have. At the moment we don't know Bob's species, so let's clear that up with a `Human` subclass:

여기서 “Bob”이라는 이름을 가지고 ‘grunt’라고 말을 하는 클래스를 만들었다. 지금까지 클래스 정의를 하고, 정의된 클래스를 이용하여 인스턴스를 만들어 보았다. 우리는 좀 더 개선해야 할 여지가 있는 것 같다. 아 잠깐! “Bob”이라는 동물이 어떤 종인지 모르니, 하위 클래스로 “Human”이라는 클래스를 만들어보자.

```
Ext.define('Human',{
    extend:'Animal',

    speak:function(){
        alert(this.getName());
    }
});
```

Now we've got a new class that inherits from Animal, therefore gaining all of its functions and configurations. We actually overrode the speak function because most humans are smart enough to say their name instead of grunt. Now, let's make Bob a human:

와~우리는 지금 새로운 클래스를 얻었다. 따라서 상위 클래스인 “Animal” 클래스에서 상속되어 모든 기능(functions)과 설정(configurations)들을 획득하였다. 그러나 실제로 대부분의 모든 사람들은 자신의 이름 정도는 말할 수 있다. 그렇기 때문에 상위 클래스의 “speak” 함수를 오버로드 하여 자신의 이름을 말하도록 변경하였다. 자. 이제 “Bob” 이라는 사람을 만들어 보자.

```
var bob = Ext.create('Human',{
    name:'Bob'
});

bob.speak();//alerts 'Bob'
```

We used a magical function when adding the Human subclass. You'll notice that we didn't actually define a getName function on our Animal class, so where did it come from? Part of the class system is the ability to give classes configuration options, which each automatically give you the following:

아 잠깐. “Human” 하위 클래스를 작성할 때 마법의 함수를 사용하였다. “Animal” 클래스, “Human” 클래스 어디에서도 “getName()”을 정의하지 않았는데, 대체 이 함수는 어디서 나온걸까? 이것은 클래스 시스템의 구성옵션의 기능의 일부이고, 아래의 항목에 해당되는 기능을 클래스 시스템에서 자동으로 추가해 준다.

- a getter function that returns the current value, in this case `getName()`.
- a setter function that sets a new value, in this case `setName()`.
- an applier function called by the setter that lets you run a function when a configuration changes, in this case `applyName()`.

- “getter function”는 지금 가지고 있는 변수의 값을 반환한다. 지금의 경우에는 “getName()” 함수이다..
- “setter function”는 새로운 값을 설정한다. 지금의 경우에는 “setName()” 함수이다.
- **an applier function called by the setter that lets you run a function when a configuration changes, in this case `applyName()`.**

The getter and setter functions are generated for free and are the recommended way to store data in a class. Every component in Sencha Touch uses the class system and the generated functions always follow the same pattern so if you know a config you already know how to get and set its value.

getter 및 setter 함수들은 별 수고없이 생성되고 클래스에 데이터를 저장하기 위해 권장되는 방법이다. Sencha Touch 모든 컴포넌트는 클래스 시스템을 사용한다. 여러분이 이미 해당 값을 설정하고 얻는 방법을 알고 있으면 생성된 함수는 항상 동일한 패턴을 따른다.

It also makes your code cleaner. For example, if you wanted to always ask the user if she really wants to change Bob's name, you can just define an `applyName` function that will automatically be called:

또한 코드를 깔끔하게 만든다. 예를 들어 “Bob”의 이름을 변경하고자 하는 경우 사용자에게 항상 답변을 얻고자 할 때, 그냥 “applyName()” 함수를 정의하여 자동으로 호출되도록 함수를 정의할 있다.

```
Ext.define('Human',{
    extend:'Animal',

    applyName:function(newName, oldName){
        return confirm('Are you sure you want to change
name to '+ newName +'?')? newName : oldName;
    }
});
```

We're just using the browser's built in confirm function, which opens a dialog asking the user the question and offering "Yes" and "No" as answers. The applier functions allow you to cancel the name change if you return false. As it happens the confirm function will return either new or old name depending on whether the user clicks Yes or No.

위 코드는 브라우저에 내장된 “confirm” 기능을 사용하여, 사용자에게 질문을 하고 “예”와 “아니오”를 선택할 수 있는 대화상자를 연다. “applier function”에서 “false” 값을 반환한 경우 이름 변경을 취소할 수 있다.

여기서 “confirm function” 은 사용자가 선택한 “예” 또는 “아니오”에 따라 신규 또는 이전 이름을 반환 한다.

If we make a new Bob and try to change his name, but then click No when prompted, his name won't change after all:

우리는 새로운 “Bob”를 만들고 그의 이름을 변경하려고 시도하였다. 그러나 프롬프트에서 “아니오”를 선택하였다. 그후 그의 이름은 변경되지 않았다.

```
var bob =Ext.create('Person',{
    name:'Bob'
});

//opens a confirm box, but we click No
bob.setName('Fred');
//still alerts 'Bob'
bob.speak();
```

We've basically already learned the important parts of classes, as follows:

지금까지 우리는 클래스의 중요한 것들을 배웠다. 아래의 항목을 보며 정리해 보도록 하자.

- All classes are defined using Ext.define, including your own classes
- Most classes extend other classes, using the `extend` syntax
- Classes are created using Ext.create, for example `Ext.create('SomeClass', {some: 'configuration'})`
- Always usine the `config` syntax to get automatic getters and setters and have a much cleaner codebase

- 모든 클래스는 “Ext.define”을 사용하여 자신만의 클래스를 정의할 수 있다.
- 대부분의 클래스들은 “extend” 구문을 사용하여 다른 클래스로 확장되었다.
- 클래스 생성은 “Ext.create”를 사용한다.
  예) Ext.create(‘SomeClass”, { some: ‘configuration” })
- 항상 “config” 구문을 사용하여 자동으로 생성되는 “getter”와 “setter”를 얻고 깔끔한 코드를 유지하는데 좀 더 힘쓰라. (??)

At this point you can already go about creating classes in your app, but the class system takes care of a few more things that will be helpful to lear are a few other things the class system does.

이 시점에서 이미 응용프로그램에 클래스를 만드는 것은 가능하다. 그러나 여러분이 클래스 시스템이 담당하는 도움이 될 만한 몇 가지 다른 것이 있다.

## Dependencies and Dynamic Loading

Most of the time, classes depend on other classes. The Human class above depends on the Animal class because it extends it - we depend on Animal being present to be able to define Human. Sometimes you'll make use of other classes inside a class, so you need to guarantee that those classes are on the page. Do this with the `requires` syntax:

```
Ext.define('Human',{
    extend:'Animal',

    requires:'Ext.MessageBox',

    speak:function(){
        Ext.Msg.alert(this.getName(),"Speaks...");
    }
});
```

When you create a class in this way, Sencha Touch checks to see if `Ext.MessageBox` is already loaded and if not, loads the required class file immediately with AJAX.

`Ext.MessageBox` itself may also have classes it depends on, which are then also loaded automatically in the background. Once all the required classes are loaded, the Human class is defined and you can start using `Ext.create` to instantiate people. This works well in development mode as it means you don't have to manage the loading of all your scripts yourself, but not as well in production because loading files one by one over an internet connection is rather slow.

In production, we really want to load just one JavaScript file, ideally containing only the classes that our application actually uses. This is really easy in Sencha Touch 2 using the JSBuilder tool, which analyzes your app and creates a single file build that contains all of your classes and only the framework classes your app actually uses. See the Building guide for details on how to use the JSBuilder.

Each approach has its own pros and cons, but can we have the good parts of both without the bad, too? The answer is yes, and we've implemented the solution in Sencha Touch 2.

## Naming Conventions

Using consistent naming conventions throughout your code base for classes, namespaces and filenames helps keep your code organized, structured, and readable.

### 1) Classes

Class names may only contain **alphanumeric** characters. Numbers are permitted but are discouraged in most cases, unless they belong to a technical term. Do not use underscores, hyphens, or any other nonalphanumeric character. For example:

- `MyCompany.useful_util.Debug_Toolbar` is discouraged
- `MyCompany.util.Base64` is acceptable

Class names should be grouped into packages where appropriate and properly namespaced using object property dot notation ( `.` ). At the minimum, there should be one unique top-level namespace followed by the class name. For example:

```
MyCompany.data.CoolProxyMyCompany.Application
```

The top-level namespaces and the actual class names should be in CamelCase, everything else should be all lower-cased. For example:

`MyCompany.form.action.AutoLoad`
Classes that are not distributed by Sencha should never use `Ext` as the top-level namespace.

Acronyms should also follow CamelCase convention, for example:

- `Ext.data.JsonProxy` instead of `Ext.data.JSONProxy`
- `MyCompany.util.HtmlParser` instead of `MyCompary.parser.HTMLParser`
- `MyCompany.server.Http` instead of `MyCompany.server.HTTP`

## 2) Source Files

The names of the classes map directly to the file paths in which they are stored. As a result, there must only be one class per file. For example:

- `Ext.mixin.Observable` is stored in `path/to/src/Ext/mixin/Observable.js`
- `Ext.form.action.Submit` is stored in `path/to/src/Ext/form/action/Submit.js`
- `MyCompany.chart.axis.Numeric` is stored in `path/to/src/MyCompany/chart/axis/Numeric.js`

`path/to/src` is the directory of your application's classes. All classes should stay under this common root and should be properly namespaced for the best development, maintenance, and deployment experience.

## 3) Methods and Variables

Similarly to class names, method and variable names may only contain **alphanumeric** characters. Numbers are permitted but are discouraged in most cases, unless they belong to a technical term. Do not use underscores, hyphens, or any other nonalphanumeric character.

Method and variable names should always use CamelCase. This also applies to acronyms.

Here are a few examples:

- Acceptable method names:
  `encodeUsingMd5()`

- `getHtml() instead of getHTML()`
- `getJsonResponse() instead of getJSONResponse()`
- `parseXmlContent() instead of parseXMLContent()`
- Acceptable variable names:
  `var isGoodName`

- `var base64Encoder`

- `var xmlReader`
- `var httpServer`

### 4) Properties

Class property names follow the same convention as method and variable names, except the case when they are static constants. Static class properties that are constants should be all upper-cased, for example:

- `Ext.MessageBox.YES = "Yes"`
- `Ext.MessageBox.NO = "No"`
- `MyCompany.alien.Math.PI = "4.13"`

# Working with classes in Sencha Touch 2.0

## 1. Declaration

### 1.1. The Old Way

If you've developed with Sencha Touch 1.x, you are certainly familiar with `Ext.extend` to create a class:

```
varMyPanel=Ext.extend(Object,{...});
```

This approach is easy to follow when creating a new class that inherits from another. Other than direct inheritance, however, there wasn't a fluent API for other aspects of class creation, such as configuration, statics, and mixins. We will be reviewing these items in detail shortly.

Let's take a look at another example:

```
My.cool.Panel=Ext.extend(Ext.Panel,{...});
```

In this example we want to namespace our new class and make it extend from `Ext.Panel`. There are two concerns we need to address:

1. `My.cool` needs to be an existing object before we can assign `Panel` as its property.
2. `Ext.Panel` needs to exist/be loaded on the page before it can be referenced.

The first item is usually solved with `Ext.namespace` (aliased by `Ext.ns`). This method recursively traverses through the object/property tree and creates them if they don't exist yet. The boring part is you need to remember adding them above `Ext.extend` all the time, like this:

```
Ext.ns('My.cool');My.cool.Panel=Ext.extend(Ext.Panel,
{...});
```

The second issue, however, is not easy to address because `Ext.Panel` might depend on many other classes that it directly/indirectly inherits from, and in turn, these dependencies might depend on other classes to exist. For that reason, applications written before Sencha

Touch 2 usually include the whole library in the form of `ext-all.js` even though they might only need a small portion of the framework.

### 1.2. The New Way

Sencha Touch 2 eliminates all those drawbacks with just one single method you need to remember for class creation: `Ext.define`. Its basic syntax is as follows:

```
Ext.define(className, members, onClassCreated);
```

Let's look at each part of this:

- `className` is the class name
- `members` is an object represents a collection of class members in key-value pairs
- `onClassCreated` is an optional function callback to be invoked when all dependencies of this class are ready, and the class itself is fully created. Due to the new asynchronous nature of class creation, this callback can be useful in many situations. These will be discussed further in Section IV.

**Example**

```
Ext.define('My.sample.Person',{
    name:'Unknown',

    constructor:function(name){
        if(name){
            this.name = name;
        }
    },

    eat:function(foodType){
        alert(this.name +" is eating: "+ foodType);
    }});var aaron
=Ext.create('My.sample.Person','Aaron');
    aaron.eat("Salad");// alert("Aaron is eating:
Salad");
```

Note we created a new instance of `My.sample.Person` using the `Ext.create()` method. We could have used the new keyword (new `My.sample.Person()`). However it is recommended that you always use `Ext.create` since it allows you to take advantage of dynamic loading. For more info on dynamic loading see the Getting Started guide

## 2. Configuration

In Sencha Touch 2, we introduce a dedicated `config` property that is processed by the powerful `Ext.Class` preprocessors before the class is created. Features include:

- Configurations are completely encapsulated from other class members.
- Getter and setter, methods for every config property are automatically generated into the class prototype during class creation if the class does not have these methods already defined.
- An `apply` method is also generated for every config property. The auto-generated setter method calls the `apply` method internally before setting the value. Override the `apply` method for a config property if you need to run custom logic before setting the value. If `apply` does not return a value then the setter will not set the value. For an example see `applyTitle` below.

Here's an example:

```
Ext.define('My.own.WindowBottomBar',
{});Ext.define('My.own.Window',{

    /** @readonly */
    isWindow:true,

    config:{
        title:'Title Here',

        bottomBar:{
            enabled:true,
            height:50,
            resizable:false
        }
    },

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

    applyTitle:function(title){
        if(!Ext.isString(title)|| title.length ===0){
            console.log('Error: Title must be a valid
non-empty string');
        }
        else{
            return title;
        }
    },

    applyBottomBar:function(bottomBar){
        if(bottomBar && bottomBar.enabled){
            if(!this.bottomBar){
returnExt.create('My.own.WindowBottomBar', bottomBar);
            }
            else{
                this.bottomBar.setConfig(bottomBar);
            }
        }
    }});
```

And here's an example of how it can be used:

```
var myWindow =Ext.create('My.own.Window',{
    title:'Hello World',
    bottomBar:{
        height:60
    }});

// logs "Hello World"
console.log(myWindow.getTitle());

myWindow.setTitle('Something New');
// logs "Something New"
console.log(myWindow.getTitle());

// logs "Error: Title must be a valid non-empty string"
myWindow.setTitle(null);

// Bottom bar's height is changed to 100
myWindow.setBottomBar({ height:100});
```

## 3. Statics

Static members can be defined using the `statics` config, as follows:

Static 맴버 또한 “statics” config를 사용하여 정의할 수 있다. 자 아래 코드를 보자.

```
Ext.define('Computer',{
    statics:{
        instanceCount:0,
        factory:function(brand){
            // 'this' in static methods refer to the 
class itself
            return new this({brand: brand});
        }
    },

    config:{
        brand:null
    },

    constructor:function(config){
        this.initConfig(config);
        // the 'self' property of an instance refers to 
its class
        this.self.instanceCount ++;
    }
});

var dellComputer =Computer.factory('Dell');
var appleComputer =Computer.factory('Mac');

// using the auto-generated getter to get the value of a 
config property. Alerts "Mac"
alert(appleComputer.getBrand());

alert(Computer.instanceCount);// Alerts "2"
```

## Error Handling and debugging

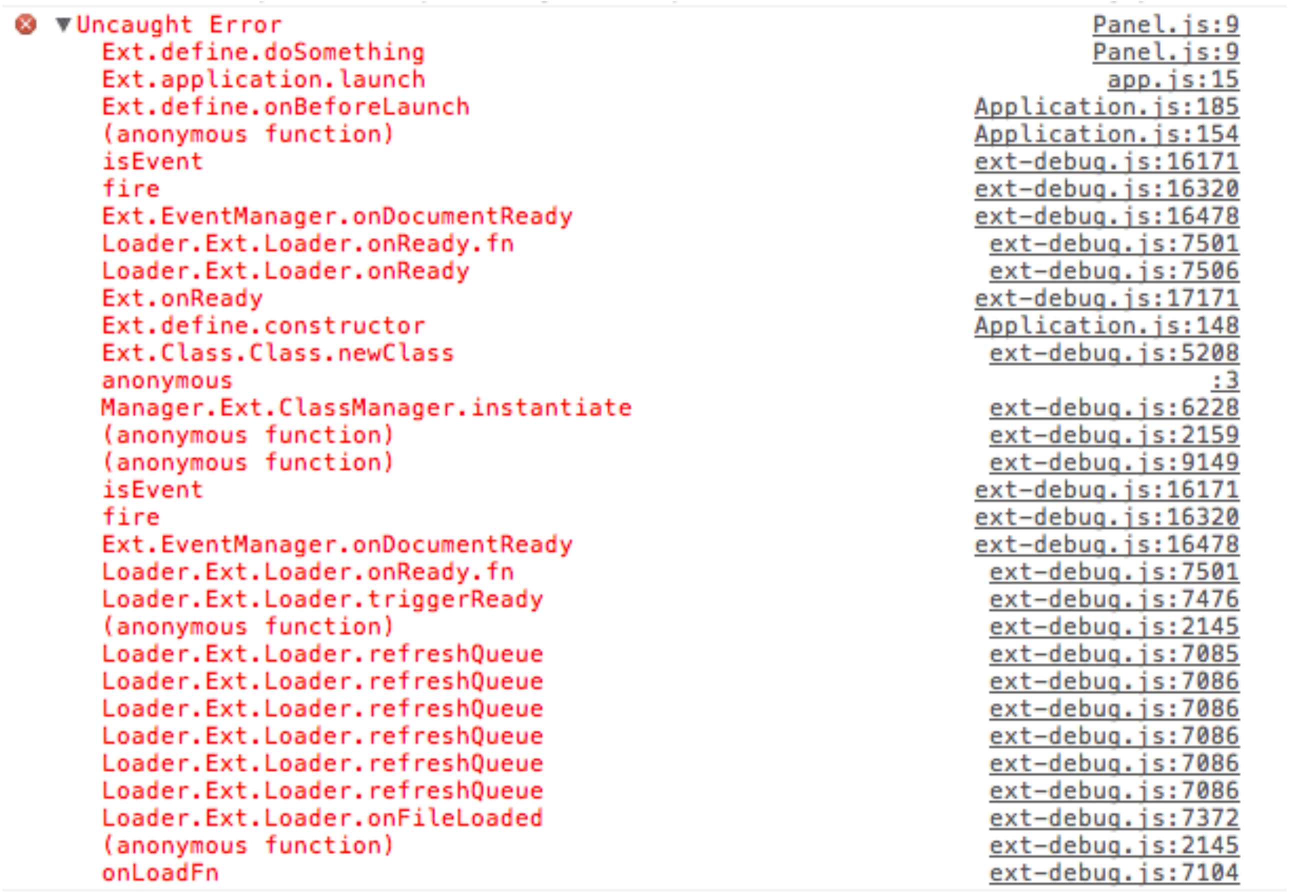
Sencha Touch 2 includes some useful features that will help you with debugging and error handling.

Sencha Touch 2에는 디버깅과 에러 처리에 도움을 줄 수 있는 유용한 기능들이 부분적으로 포함되어 있다.

You can use `Ext.getDisplayName()` to get the display name of any method. This is especially useful for throwing errors that have the class name and method name in their description, such as:

```
throw new Error('['+Ext.getDisplayName(arguments.callee)
+'] Some message here');
```

When an error is thrown in any method of any class defined using `Ext.define()`, you should see the method and class names in the call stack if you are using a WebKit based browser (Chrome or Safari). For example, here is what it would look like in Chrome:



```
▼Uncaught Error                                              Panel.js:9
   Ext.define.doSomething                                    Panel.js:9
   Ext.application.launch                                      app.js:15
   Ext.define.onBeforeLaunch                         Application.js:185
   (anonymous function)                              Application.js:154
   isEvent                                           ext-debug.js:16171
   fire                                              ext-debug.js:16320
   Ext.EventManager.onDocumentReady                  ext-debug.js:16478
   Loader.Ext.Loader.onReady.fn                       ext-debug.js:7501
   Loader.Ext.Loader.onReady                          ext-debug.js:7506
   Ext.onReady                                       ext-debug.js:17171
   Ext.define.constructor                            Application.js:148
   Ext.Class.Class.newClass                           ext-debug.js:5208
   anonymous                                                         :3
   Manager.Ext.ClassManager.instantiate               ext-debug.js:6228
   (anonymous function)                               ext-debug.js:2159
   (anonymous function)                               ext-debug.js:9149
   isEvent                                           ext-debug.js:16171
   fire                                              ext-debug.js:16320
   Ext.EventManager.onDocumentReady                  ext-debug.js:16478
   Loader.Ext.Loader.onReady.fn                       ext-debug.js:7501
   Loader.Ext.Loader.triggerReady                     ext-debug.js:7476
   (anonymous function)                               ext-debug.js:2145
   Loader.Ext.Loader.refreshQueue                     ext-debug.js:7085
   Loader.Ext.Loader.refreshQueue                     ext-debug.js:7086
   Loader.Ext.Loader.refreshQueue                     ext-debug.js:7086
   Loader.Ext.Loader.refreshQueue                     ext-debug.js:7086
   Loader.Ext.Loader.refreshQueue                     ext-debug.js:7086
   Loader.Ext.Loader.refreshQueue                     ext-debug.js:7086
   Loader.Ext.Loader.onFileLoaded                     ext-debug.js:7372
   (anonymous function)                               ext-debug.js:2145
   onLoadFn                                           ext-debug.js:7104
```

## Further Reading

Classes are just part of the Sencha Touch 2 ecosystem. To understand more about the framework and how it works, we recommend the following:

클래스 시스템은 Sencha Touch 2 프레임워크의 생테계의 일부분이다. 프레임워크와 활용하는 방법에 대해서 좀 더 자세히 알고 싶다면 아래의 글을 읽어보기 바란다.

- Components and Containers
- The Data Package
- The Layout System
- Getting Started