**What is Prototype ?**

* Prototype is a JavaScript Framework that aims to ease the development of dynamic web applications
* Extends DOM elements and built-in types with useful methods.
* Has built-in support for class-style OOP including inheritance
* Ajax features

**Download Prototype**

Go to the download page (<http://prototypejs.org/download/>) and download the latest version

**Defining classes and inheritance**

* To define a class in we use **Class.create** method
* **Initialize** is a function which gets called once you create an instance of the class, so if you want to pass any parameters which you want to be globally available in the class pass it to the class while you create it.
* The basic example to create a class would
  + **var** Person **=** Class.create({  
     initialize**:** **function**(name) {  
     **this**.name **=** name;  
     },  
     say**:** **function**(message) {  
     **return** **this**.name **+** ': ' **+** message;  
     }  
    });
* If you want to create a subclass of **Person,** again use **Class.create** method and specify the class you want to inherit from as the first parameter and then the methods you want to override
  + **var** Pirate **=** Class.create(Person, {  
     *// redefine the speak method*  
     say**:** **function**($super, message) {  
     **return** $super(message) **+** ', yarr!';  
     }  
    });  
       
    **var** john **=** **new** Pirate('Long John');  
    john.say('ahoy matey');
  + When you override a method in a subclass, but still want to be able to call the original method, you will need a reference to it.
  + You will use **$super** for the referencing it.
  + Prototype will detect this and make the overridden method available to you through that argument
  + Even though the **say** method has two parameters now, you only need to pass one argument while calling it.

**Defining modules and using it while creating class (mix-in modules)**

* You can define a module
  + var module = {  
     validate : function(username, password) {  
     if(this.username != username || this.password != password) {  
     alert('Invalid login details. Please try again');  
     } else {  
     alert('Login successfull');  
     }  
     }  
     };
  + Now using the above created module you can create a class
  + var ValidateClass = Class.create(module, {  
     initialize : function() {  
     this.username = 'admin';  
     this.password = 'password';  
     }  
     });  
    var validateObj = new ValidateClass();  
    validateObj.validate('admin', 'passwordasd');

**Objects are copied by reference, not by value**

Image an class which has a property which gets updated by a method in the class itself.

var Logger = Class.create({  
 initialize : function() { },

log: [],  
 write : function(message) {  
 this.log.push(message);  
 }  
 });

var logger = new Logger();  
logger.write('India');  
logger.write(US); alert(logger.log);

The above example looks fine. But what happens if we create another instance of the class.

var logger2 = new Logger();  
alert(logger.log);

Even though we expected an empty array we still get the value which got updated by the previous instance of the class. This is because all the instance share the same array object because it is copied by reference, not by value.

Instead, **initialize your instances with default values**

**var** Logger **=** Class.create({  
 initialize**:** **function**() {  
 *// this is the right way to do it:*  
 **this**.log **=** [];  
 },  
 write**:** **function**(message) {  
 **this**.log.push(message);  
 }  
});

**Defining class methods**

* Simply define them on your existing classes
  + validateObj.validateUsername = function(username) {

if(username.length <= 10) {

alert('Your username should be atleast 10 characters long.')

}

};

* If you want to define multiple methods at once, you could use Object.extend
  + Object.extend(validateObj, {  
     sayHi: function() {  
     alert('Hi everyone...');  
     },  
     sayHello : function() {  
     alert('Hello everyone...');  
     }  
    });

Note : Class methods you add do not get inherited.

**Class properties**

* Superclass
* Subclasses

Person.superclass  
*// -> null*  
Person.subclasses.length  
*// -> 1*  
Person.subclasses.first() **==** Pirate

**Class#addMethods**

Using addMethods(), you can dynamically add extra methods to already defined class.

Person.addMethods({  
 sayHi:function() {  
 alert('Hi Person method, new method added !!!')

}  
});

**DOM**

* Most of the DOM methods are encapsulated by the **Element.Methods**
* Prototype adds many convenience methods to elements returned by the **$() function**
* **Adding your own methods with Element.addMethods**
  + var element\_methods\_obj = {  
     append\_txt : function(element, text\_to\_append) {  
     element = $(element);  
     element.update(element.innerHTML+', '+text\_to\_append);  
     }  
    };  
    Element.addMethods(element\_methods\_obj);  
    $('abc').append\_txt('This is my first Element function, created using prototypejs');

**Ajax**

* Ajax functionality is contained in the global Ajax object
* Actual requests are made by creating instances of the Ajax.Request object.
* Syntax
  + new Ajax.Request('/url', { options});
* Callback function to capture response
  + onSuccess
  + onFailure
* Example for a making an ajax request would be
  + new Ajax.Request('http://localhost/training/prototypejs/remote\_file.php', {  
     method:'get',  
     onSuccess: function(transport) {  
     document.getElementById('abc').innerHTML = transport.responseText;  
     },  
     onFailure: function() {  
     alert('Error occurred while requesting remote data');  
     }  
    });
  + Method can be either get or post
  + You can also pass data to the server using **parameters** property
  + **new** Ajax.Request('/some\_url', {  
     method**:** 'get',  
     parameters**:** {company**:** 'example', limit**:** 12}  
    });
* **Responders**
  + Using Ajax.Responders you can register callbacks that will fire on a certain state of any ajax request made.
  + Ajax.Responders.register({  
     onCreate**:** **function**(){  
     alert('a request has been initialized!');  
     },   
     onComplete**:** **function**(){  
     alert('a request completed');  
     }  
    });
* **Updaters**
  + Using Ajax.Updater you can make Ajax requests to receive HTML fragments that update parts of the document.
  + **new** Ajax.Updater('products', '/some\_url', {  
     method**:** 'get',  
     insertion**:** Insertion.Top  
    });
  + Using Insertion.Top or Insertion.Bottom you can insert the contents on top or bottom of the container without clear the data from the previous request

* **PeriodicalUpdater**
  + With Ajax.PeriodicalUpdater you make ajax request in periodic intervals to repeatedly fetch contents from the server.
  + **new** Ajax.PeriodicalUpdater('products', '/some\_url', {  
     method**:** 'get',  
     insertion**:** Insertion.Top,  
     frequency**:** 1,  
     decay**:** 2  
    });
  + Frequency is the interval at which the request is made
  + Decay is the factor by which the frequency is multiplied every time when current response body is the same as previous one
  + If the server content keeps changing in every request, then decay does not take effect.

**JSON**

* Object.toJSON()
  + Pass object to serialized
  + Returns JSON string
  + var data = {name: 'Violet', occupation: 'character', age: 25};  
     var json\_obj = Object.toJSON(data);  
     console.log(json\_obj )
* evalJSON()
  + var data = '{ "name": "Violet", "occupation": "character" }'.evalJSON();  
    Data.name;

**Event delegation**

* Event delegation is an advanced technique for event-driven programming
* Instead of attaching one listener onto each element in a group, attach one listener onto an ancester shared by all the elements in that group
* When the listener is triggered, determine whether to act by looking at which element originally received the action.
* Consider the example below
* <div id="items">  
   <ul>  
   <li>lorem</li>  
   <li>ipsum</li>  
   <li>dolor</li>  
   </ul>  
  </div>
* $$('#items li').each( **function**(item) {  
   item.observe('click', **function**(event) {  
   doSomethingWith(event.target);  
   });  
  });
* Issue with the above approach is that, if the list is huge, attaching many event listeners will affect performance,
* And we need to update the content dynamically, it requires to reattach all the event listeners
* Instead lets try the code below
  + $('items').observe('click', **function**(event) {  
     **if** (event.target.tagName **===** 'LI') {  
     doSomethingWith(event.target);  
     }  
    });
* Now in the latest version of Prototype, extracted this idea into a pattern.
  + $('items').on('click', 'li', **function**(event, element) {  
     doSomethingWith(event.element);  
    });