FRONTEND DEVELOPMENT

Advance JavaScript

ACADGILD

Introduction to Object Oriented

- Object-oriented programming allows you to reuse code without having to copy or recreate it

- Many popular programming languages (such as Java, JavaScript, C#, C++, Python, PHP, Ruby and Objective-C) support object-oriented programming (OOP).

Terminology

Namespace

A container which allows developers to bundle all functionality under a unique, application-specific name.

Property

An object characteristic, such as color.

Method

An object capability, such as walk. It is a subroutine or function associated with a class.

Constructor

A method called at the moment of instantiation of an object. It usually has the same name as that of the class containing it.

Class

Defines the characteristics of the object. It is a template definition of variables and methods of an object.

Object

An Instance of a class.

Inheritance

A class can inherit characteristics from another class.

Encapsulation

A method of bundling the data and methods that use them together.

Abstraction

The conjunction of complex inheritance, methods, properties of an object must be able to simulate a reality model.

Polymorphism

Poly means "many" and morphism means "forms". Different classes might define the same method or property.

Functions

- A JavaScript function is a block of code designed to perform a particular task.
- Define the code once, and use it many times.
- A JavaScript function is executed when "something" invokes it (calls it).
- **Syntax**

```
function name(parameter1, parameter2, parameter3) {
    code to be executed

   return value;
}
```

Anonymous Function

- Sometimes you need a simple function without the need of assigning it to a name. This is called a anonymous function
- **Example**

```
var nums = [1, 4, 3, 2, 6, 2, 0];
nums.sort( function(a,b){ return a-b; } );
```

Immediate Function

- An immediate function is a function that executes as soon as it is defined.

```
var myName = ‘Acadgild‘;

(function(thisName){
   console.log( 'hello, my name is: ' + thisName );
}(myName))
```

Inner Functions

- Nesting functions within functions.

```
function Ftimes() {
  var FtimesObj = new Object()
        function Ftimes3(x) {
          return x * 3
          }
        function Ftimes4(x) {
          return x * 4
          }
  FtimesObj.Ftimes3 = Ftimes3
  FtimesObj.Ftimes4 = Ftimes4
  return FtimesObj
  }

Multi = new Ftimes()
alert(Multi.Ftimes3(5)) // alerts 15
```

Closures

- A closure takes place when a function creates an environment that binds local variables to it in such a way that they are kept alive after the function has returned.
- A closure is a special kind of object that combines two things: a function, and any local variables that were in-scope at the time that the closure was created.

Closure Example

```
function getNameFunction() {

        var name = “Acadgild";

        return function getName() { return name; }
        }

        var displayName = function() {
        var getName = getNameFunction();
        alert( "Hello " + getName() + "!" );
         return getName;
        }(); //holds the getName() function

alert(displayName); //call it again later…

setTimeout( 'alert( "Your name is " + displayName() )', 10 );
```

JavaScript Object Literal

- An object literal is a comma-separated list of name-value pairs wrapped in curly braces.
- Object literals encapsulate data, enclosing it in a tidy package.
- This minimizes the use of global variables which can cause problems when combining code.
- Example

```
var myObject = {
sProp: 'some string value',
numProp: 2, bProp: false
 };
```

Creating Object using Constructor

- Sometimes we like to have an "object type" that can be used to create many objects of one type.
- The standard way to create an "object type" is to use an object constructor function:

```
function person(first, last, age, eyecolor) {
   this.firstName = first;
   this.lastName = last;
   this.age = age;
   this.eyeColor = eyecolor;
}
var myFather = new person("John", "Doe", 50, "blue");
var myMother = new person("Sally", "Rally", 48, "green");
```

Private, Privileged, Public And Static Members

```
function Kid (name) {   // Constructor
            var idol = "Paris Hilton"; // Private
            this.getIdol = function () { return idol; };  // Privileged
            this.name = name; // Public
      }
// Public
    Kid.prototype.getName = function () { return this.name; };
// Static property
     Kid.town = "South Park";
```

```
// Create a new instance
      var cartman = new Kid("Cartman");
// Access private property
      cartman.idol; // undefined
// Access privileged method
      cartman.getIdol(); // "Paris Hilton“
 // Access public property
      cartman.name; // "Cartman"
// Access public method
      cartman.getName(); // "Cartman"
// Access static property on an instance
      cartman.town; // undefined
// Access static property on the constructor object
       Kid.town; // "South Park"
```

JavaScript Object Property

- Properties are the most important part of any JavaScript object.
- Properties are the values associated with a JavaScript object.
- A JavaScript object is a collection of unordered properties.
- Properties can usually be changed, added, and deleted, but some are read only.

Accessing JavaScript Property

The syntax for accessing the property of an object is:

- objectName.property
- objectName["property"]
- objectName[expression]

Enumerable Properties

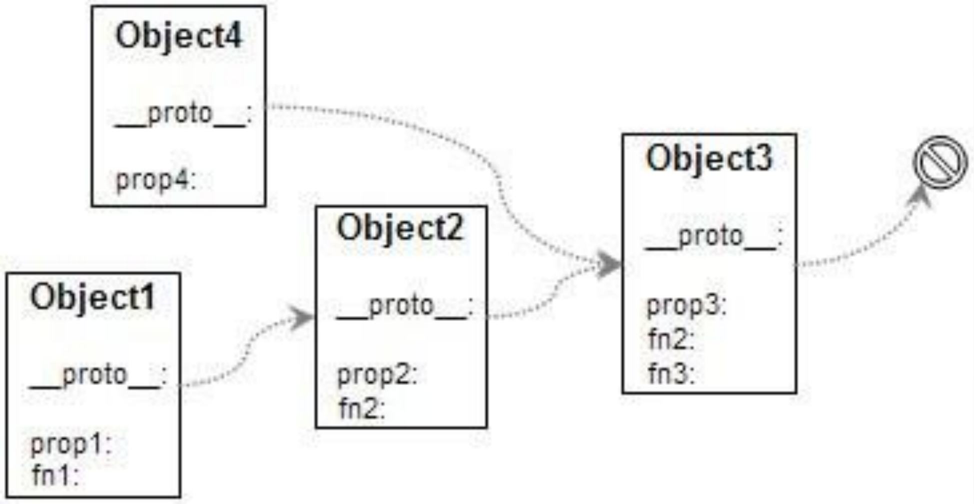
- **Enumerable :** true or false. Whether the property shows in some loop constructs,

  such as for (var x in o) {...} andObject.keys(o)

- **Checking Property's Enumerable Attribute**

  obj.propertyIsEnumerable(p).

Prototype



An ancestor of a JavaScript object

- like a "super-object" instead of a superclass
- a parent at the object level rather than at the class level

Every object contains a reference to its prototype

- default: Object.prototype; strings → String.prototype; etc.

A prototype can have a prototype, and so on

- an object "inherits" all methods/data from its prototype(s)
- doesn't have to make a copy of them; saves memory
- prototypes allow JavaScript to mimic classes, inheritance

Functions and Prototype

Every function stores a prototype object property in it

- example: when we define our Point function (constructor), that creates a Point.prototype
- initially this object has nothing in it ({})
- every object you construct will use the function's prototype object as its prototype

Every new Point object uses Point.prototype

```
// also causes Point.prototype to be defined
function Point(xValue, yValue) {
    ...
}
```

_ _proto_ _ Property (Object)

- Contains a reference to the internal prototype of the specified object.

  object._ _proto_ _

- **Parameters**

  object : Required. The object on which to set the prototype.

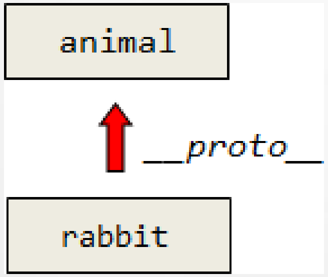
```
function Rectangle() {  }

var rec = new Rectangle();

if (console && console.log) {
        console.log(rec._ _proto_ _ === Rectangle.prototype);
        // Returns true
        rec._ _proto_ _ = Object.prototype;
        console.log(rec._ _proto_ _ === Rectangle.prototype);
        // Returns false
 }
```

Prototypal Inheritance

- In JavaScript, the inheritance is prototype-based. That means that there are no classes. Instead, an object inherits from another object
- Inheritance, the _ _proto_ _



- When an object rabbit inherits from another object animal, in JavaScript that means that there is a special property

rabbit. _ _proto_ _ = animal.

Multiple Inheritance

- Inheritance is all about copying properties from parent to child prototype, then why not copy properties from multiple parents.

```
function multiInheritance() {
        var n = {}, stuff, j = 0, length = arguments.length;
              for (j = 0; j <length; j++) {
              stuff = arguments[j];
                     for (var index in stuff) {
                                   if (stuff.hasOwnProperty(index)) {
                                   n[index] = stuff[index];
                                   }
                     }
              }
        return n;
}
```

Parasitic inheritance

- This Pattern as suggested by Douglas Crockford, is called parasitic inheritance.
- It's about a function that creates objects by taking all of the functionality from another object into a new one, augmenting the new object, and returning it.
- Parasitic inheritance is different from prototypal inheritance which we have discussed so far. Prototypal inheritance is used more often because its more efficient.

Copy Prototype of Inheritance

- Clone()
- **MyClass.prototype = clone(AnotherClass.prototype);**
- By cloning the prototype we get a new copy of it and assign that to MyClass's prototype so that changing the inherited properties will not affect the parent's prototype's properties.
- Like this would **MyClass.prototype = AnotherClass.prototype.**

```
function clone (obj) {
function CloneFactory () {}
CloneFactory.prototype = obj;
return new CloneFactory();
}
```

- **copy()** makes a shallow, non-recursive copy of a single object. This implementation is interesting because it handles native types and correctly copies objects created by a user-defined class.

- **deepCopy()** is the entry point for the deep copy algorithm. Every member is recursively deep copied.

Deep Copy

```
function deepCopy(p, c) {
        c = c || {};
        for (var index in p) {
                if (p.hasOwnProperty(index)) {
                        if (typeof p[index] === 'object') { c[index] =
                                                        Array.isArray(p[index]) ? [] : {};
                        deepCopy(p[index], c[index]);
                        }
                        else {
                        c[index] = p[index];
                        }
                }
        }
 return c;
}
```