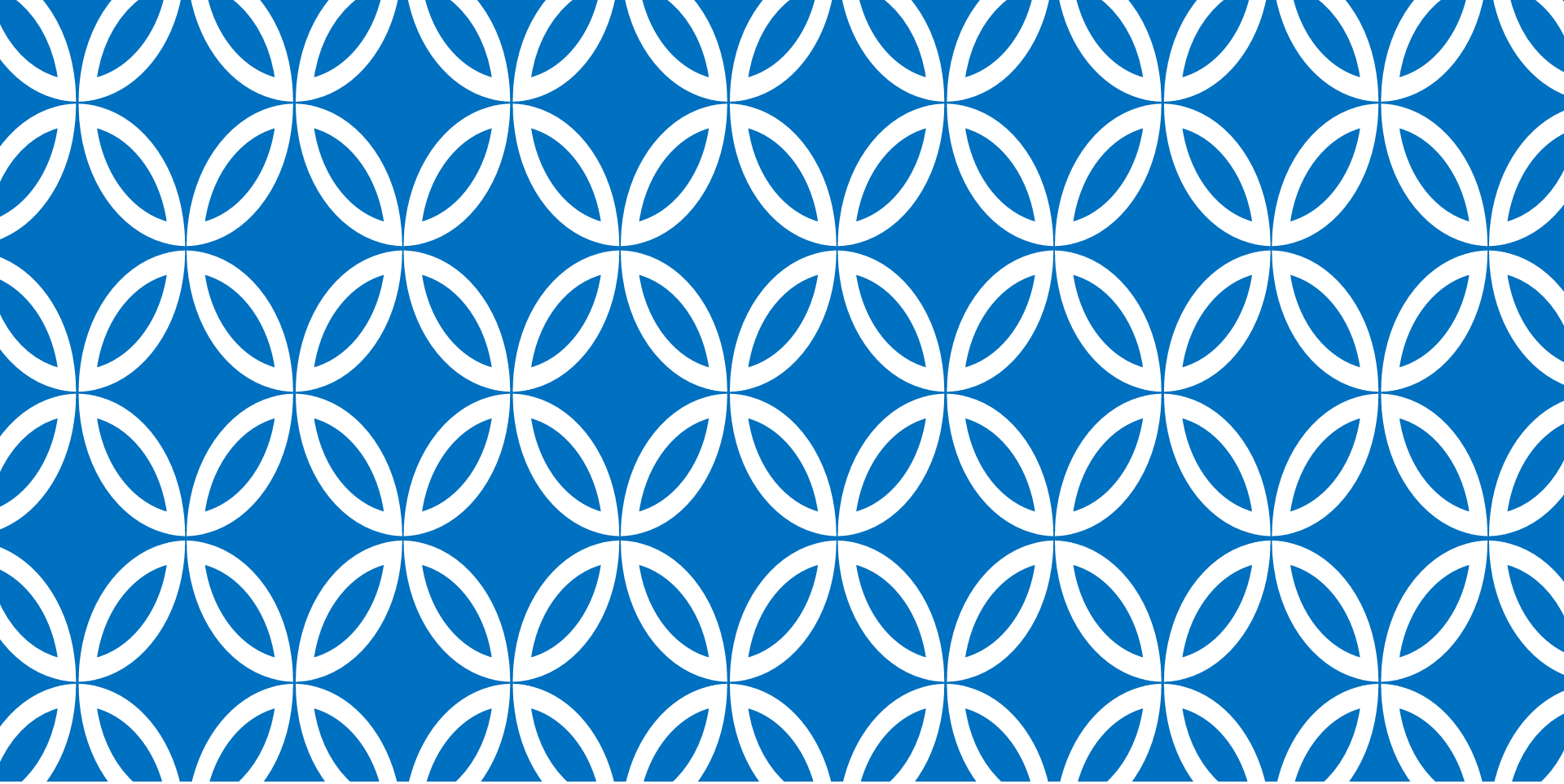

ACADGILD

# FRONT END DEVELOPMENT (WITH ANGULARJS)

Website : http://www.acadgild.com
LinkedIn : https://www.linkedin.com/company/acadgild
Facebook : https://www.facebook.com/acadgild



# Session 8 – Advanced JavaScript (contd.)

# Agenda – Advanced JavaScript

- JavaScript Object Literal
- Creating Object using Constructor
- Creating Object using existing Object
- Private, Privileged, Public & Static Members
- JavaScript Object Property
- Accessing JavaScript Property

# JavaScript Object Literal

- An object literal is a comma-separated list of name-value pairs wrapped in curly braces.
- An object literal encapsulate data, enclosing it in a tidy package. This process minimizes the use of global variables which can cause problems when combining the code.
- Object literal property values can be of any data type, including array literals, functions, and nested object literals.

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 as shown below:

```
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");
```

# Creating Object Using Existing Object

- **Object.create()** takes one object and makes it the prototype of a newly created object.
- used to create new objects using existing object

**Syntax :**

- **Object.create**(object, propertiesObject);
  - **object** – Required Argument
  - **propertiesObject** – Optional Argument

- Var newObject = Object.create(myFather);
- 'newObject' will contain same properties as 'myFather' object
- Console.log(newObject.firstName) //prints John

# Private, Privileged, Public & Static Members

- Private variables are declared with ‘var’ keyword inside the object and can be accessed only by private functions and privileged methods.
- Privileged methods are declared with *this.methodName=function(){…}* and can be called by privileged methods.
- Public properties are declared with *this.variableName* and can be read or written from outside the object.
- Static properties are defined by *Classname.propertyname = someValue*

```
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";
```

# Private, Privileged, Public & Static Members (contd.)

```
// 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.
- Properties can usually be changed, added and deleted but some are read only.
- A JavaScript object is a collection of unordered properties.

# Accessing JavaScript Property

- The syntax for accessing the property of an object is:
  - objectName.property
  - objectName["property"]
  - objectName[expression]

# Agenda – Advanced JavaScript

1. Enumerable Properties
2. Prototype
3. Functions and Prototype
4. _ _proto_ _ Property (Object)
5. Prototypal Inheritance
6. Multiple Inheritance
7. Parasitic inheritance
8. Copy Prototype of Inheritance
9. Deep Copy

# Enumerable Properties

- **Enumerable** properties are those which can be iterated by a for…in loop.
- **Enumerable:** true or false. Whether the property shows in some loop constructs, such as

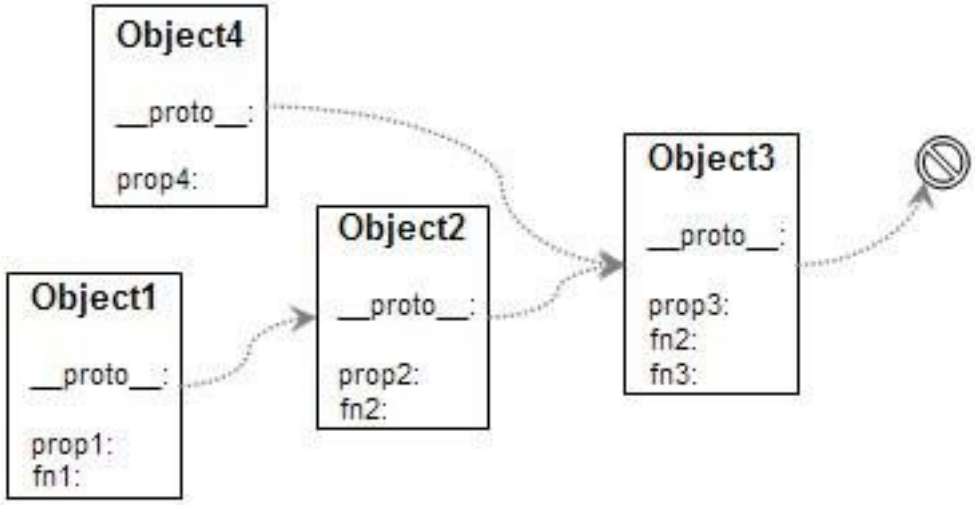
  for (var x in o) {…} and Object.keys(o)

- **Checking Property's Enumerable Attribute**

  obj.propertyIsEnumerable(p)

# Prototype

- Every JavaScript object has a prototype. The prototype is also an object.
- **An ancestor of a JavaScript object is**
  - like a "super-object" instead of a superclass
  - a parent at the object level rather than at the class level



# Prototype (contd.)

- **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.

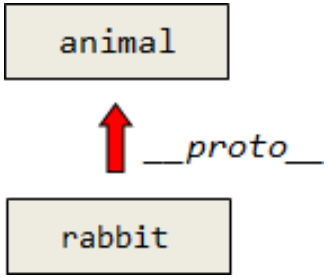
```
function Rectangle() {  }

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.**

- When a *rabbit* property is accessed and the interpreter can't find it in *rabbit*, it follows the *_proto_ link* and searches in *animal*.

# Multiple Inheritance

- Inheritance is all about copying properties from parent to child prototype, then why not copying 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.
- In parasitic inheritance a power constructor calls another constructor, takes the result, augments it and returns it as though it did all the work.
- Parasitic inheritance is different from prototypal inheritance which we have discussed so far.
- Prototypal inheritance is used more often because its more efficient than parasitic inheritance.

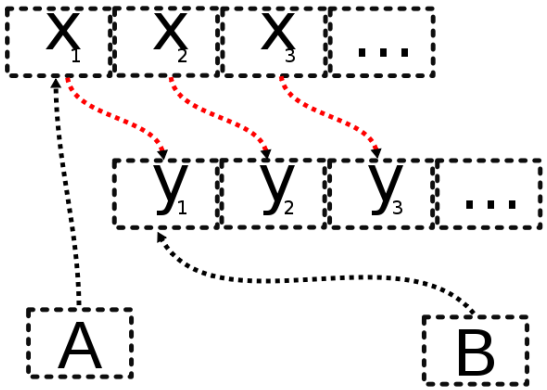
# Copy Prototype of Inheritance

- 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

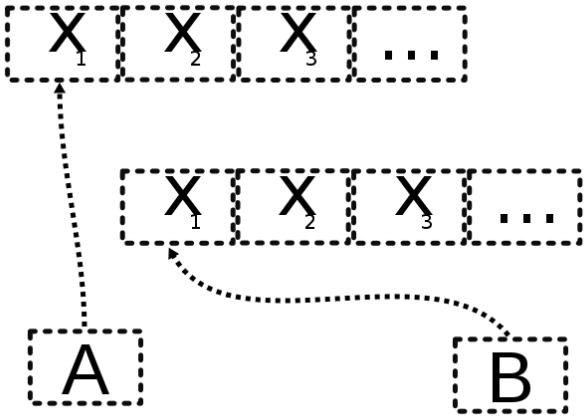
```
function clone (obj)
{
   function CloneFactory () {}
   CloneFactory.prototype = obj;
   return new CloneFactory();
}
```

# Copy Prototype of Inheritance

- **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 in Progress




Deep Copy is completed

# 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;
}
```

# Lets Discuss Assignments


*Assignment*

# Get in Touch with ACADGILD

**Contact Info:**

- **Website : http://www.acadgild.com**
- **LinkedIn : https://www.linkedin.com/company/acadgild**
- **Facebook : https://www.facebook.com/acadgild**
- **Support: support@acadgild.com**