Globant

we are ready

Front End Engineer

Core JavaScript Concepts

Topics

JavaScript Data Types
Objetcs in JavaScript
Execution Context
Variable Scope
Variable Hoisting
Scope Chain
Closures
Inheritance

DATA TYPES

A data type is a specific category of information that a variable contains.

A variable's specific data type is very important because the data type helps determine how much memory the computer will allocate for the data stored in the variable.

The data type also governs the kinds of operations that can be performed on a variable.

OBJECT

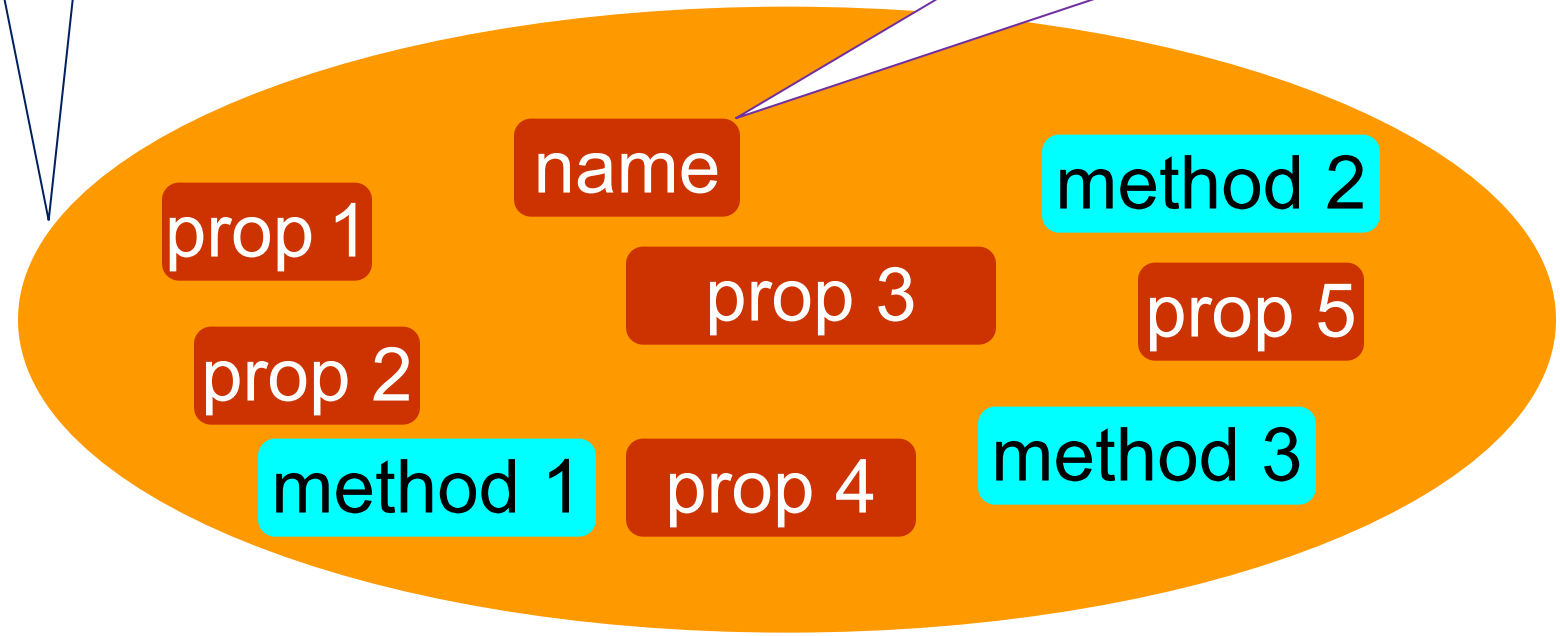
Object: A *named* collection of properties (data, state) & methods (instructions, behavior)

Everything that JavaScript manipulates, it treats as an *object* – e.g. a window or a button.
An object has *properties* – e.g. a window has size, position, status, etc.
An object can be manipulated with *methods* that are associated with that object – e.g. a resize a window with *resizeTo(150, 200)*

OBJECTS

A collection of properties & methods

All objects have the “name” property: it holds the name of the object (collection)



OBJECT-BASED NOT OBJECT-ORIENTED!

JavaScript is not a true object-oriented language like C++ or Java

It is so because it lacks two key features:
A formal inheritance mechanism
Strong typing

Nevertheless, JavaScript shares many similarities with object-oriented languages, and therefore is called an object-based language

Now, you can see:

01 JavaScript Introduction to Object Oriented Programming

http://youtu.be/cwUw0rGZC70

JAVASCRIPT DATA TYPES

Primitive Data Types:
Number
String
Boolean

Special Data Types:
Null
Undefined

Everything else is an Object
Arrays and **Functions** have some special features, but they're still objects

Now, you can see:

02 JavaScript Primitives and Objects

http://youtu.be/MuHsRavjZYk

TYPEOF

The `typeof` prefix operator returns a string identifying the type of a value.

type	typeof
object	'object'
function	'function'
array	'object'
number	'number'
string	'string'
boolean	'boolean'
null	'object'
undefined	'undefined'

TYPEOF

Be careful with the **typeof** statement!

```
typeof undefined; // undefined
typeof null; // object (THIS IS WRONG!)

typeof function(){}; // function
typeof ['a', 'b']; // object (NOT VERY HELPFUL)
```

Now, you can see:

03 JavaScript Primitive Types vs Reference Types

http://youtu.be/zVlY9bHvkd4

INHERITANCE

Inheritance is object-oriented code reuse.

Two Schools:

- Classical
- Prototypal

Objects are instances of Classes.

A Class inherits from another Class.

CLASSICAL INHERITANCE

PSEUDOCLASSICAL

Pseudoclassical looks sort of classical, but is really prototypal.

Three mechanisms:

- *Constructor* functions.
- The **`prototype`** member of functions.
- The **`new`** operator.

CONSTRUCTOR FUNCTIONS

```
function Constructor() {
    this.member = initializer;
    return this;  // optional
}
Constructor.prototype.firstMethod =
    function (a, b) {...};
Constructor.prototype.secondMethod =
    function (c) {...};

var newObject = new Constructor();
```

NEW OPERATOR

```
var newObject = new Constructor();
```

`new Constructor()` returns a new object with a link to `Constructor.prototype`.

NEW OPERATOR

The **new** operator is required when calling a Constructor.

If **new** is omitted, the global object is clobbered by the constructor, and then the global object is returned instead of a new instance.

THE PROTOTYPE PROPERTY

The prototype property is a built-in property:

Specifies the constructor from which an object was extended

When you instantiate a new object named valentinesDay based on the CandyOrder constructor function the new object includes the

customerName
candyType
numBoxes

Properties Along with the showOrder() method

THE PROTOTYPE PROPERTY

After instantiating a new object you can assign additional properties to the object, using a period.

```
var birthday = new CandyOrder("Don", "chocolate", 5);
birthday.orderDate = "June 1, 2005";
```

When you add a new property this way, the property is only available to the specific object birthday.

The property is not available to the constructor function or other objects instantiated from the same constructor function.

THE PROTOTYPE PROPERTY

If you use the prototype property with the name of the constructor function, any new properties you create will also be available to the constructor function and any object it extends to

```
var birthday = new CandyOrder("Don", "chocolate", 5);
CandyOrder.prototype.orderDate = "June 1, 2005";
```

In this case, all CandyOrder objects would have an order date of June 1, 2005

Because not all orders will take place on June 1, 2005

```
CandyOrder.prototype.orderDate = "";  // assign empty value
//then assign the order date to each individual CandyOrder object
```

SYNTACTIC RAT POISON

```
Constructor.
 method('first_method',
        function (a, b) {...}).
 method('second_method',
        function (c) {...});
```

```
Function.prototype.method =
  function (name, func) {
        this.prototype[name] = func;
        return this;
    };
```

Now, you can see:

04 JavaScript Add Object Method and Property to Class

http://youtu.be/WPYHx-e40DQ

PSEUDOCLASSICAL INHERITANCE

Classical inheritance can be simulated by assigning an object created by one constructor to the **`prototype`** member of another.

```
function BiggerConstructor() {...};
BiggerConstructor.prototype =
    new Constructor();
```

This does not work exactly like the classical model.

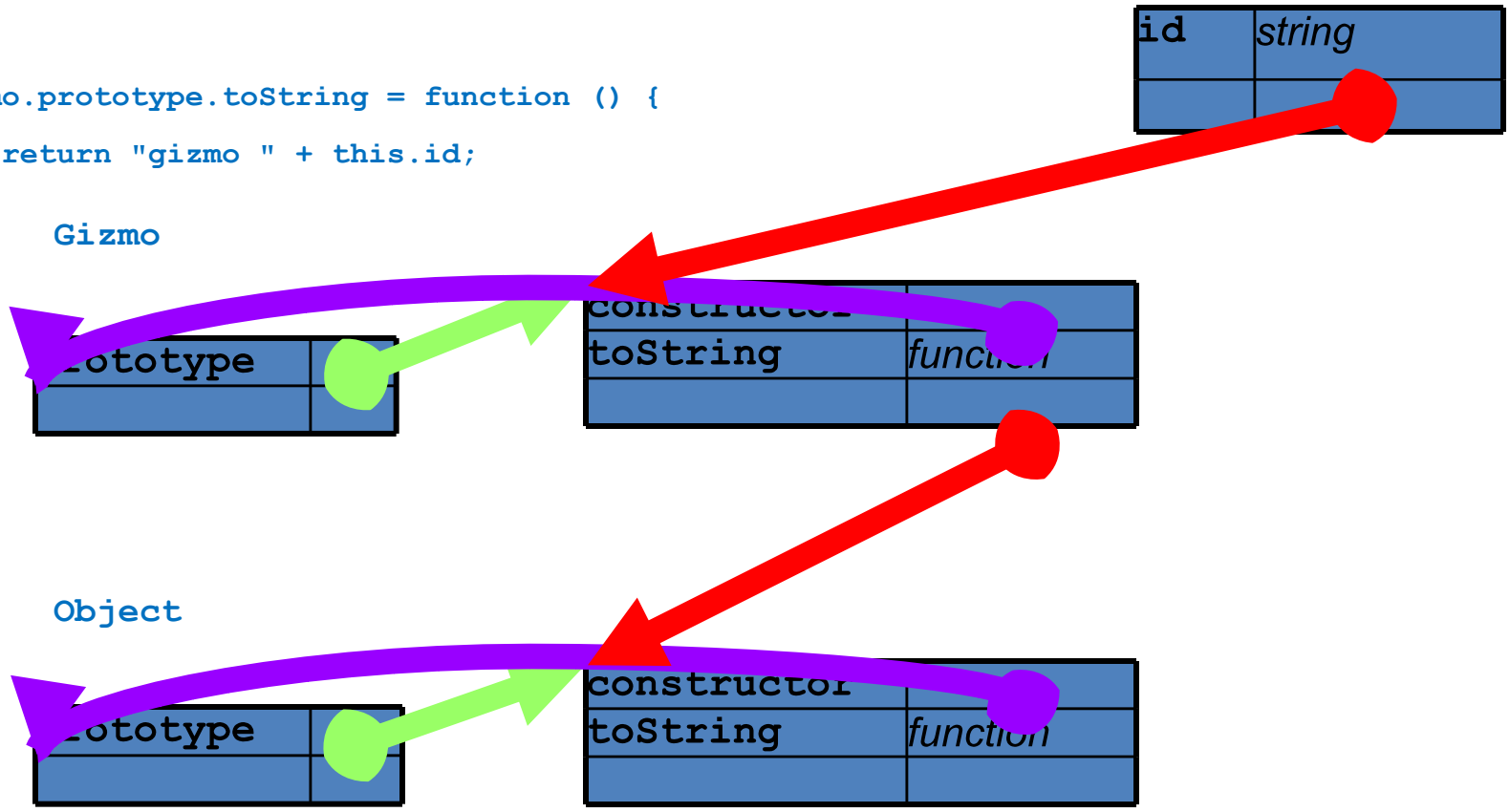
EXAMPLE

```
function Gizmo(id) {
    this.id = id;
}
Gizmo.prototype.toString = function() {
    return "gizmo " + this.id;
};
```

EXAMPLE

```
function Gizmo(id) {
    this.id = id;
}
Gizmo.prototype.toString = function () {
    return "gizmo " + this.id;
};
```

new Gizmo(*string*)



INHERITANCE

If we replace the original **`prototype`** object with an instance of an object of another class, then we can inherit another class's stuff.

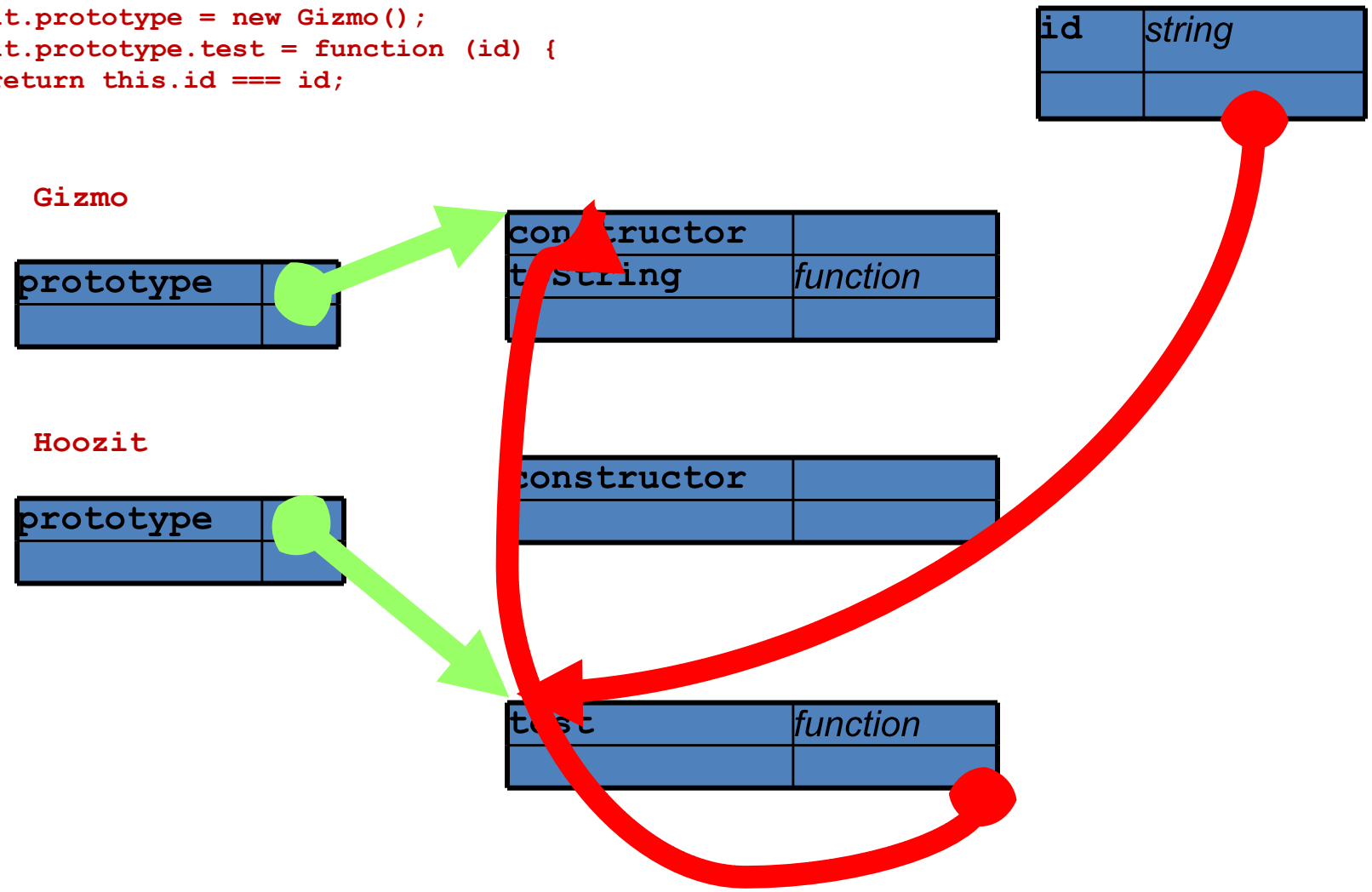
EXAMPLE

```
function Hoozit(id) {
    this.id = id;
}
Hoozit.prototype = new Gizmo();
Hoozit.prototype.test = function (id) {
    return this.id === id;
};
```

EXAMPLE

```
function Hoozit(id) {
    this.id = id;
}
Hoozit.prototype = new Gizmo();
Hoozit.prototype.test = function (id) {
    return this.id === id;
};
```

new Hoozit(*string*)



PROTOTYPAL INHERITANCE

Class-free.
Objects inherit from objects.
An object contains a secret link to the object it inherits from.

```
var newObject = object(oldObject);
```

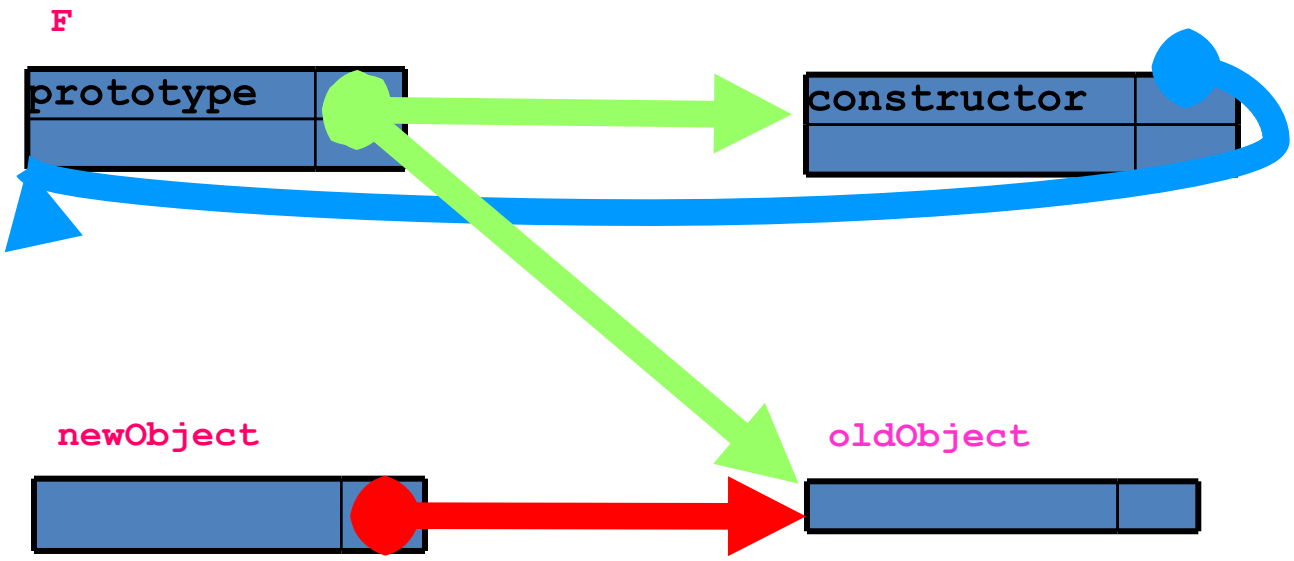
OBJECT FUNCTION

A prototypal inheritance language should have an operator like the `object` function, which makes a new object using an existing object as its prototype.

```
function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}
```

OBJECT FUNCTION

```
function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}
newObject = object(oldObject)
```

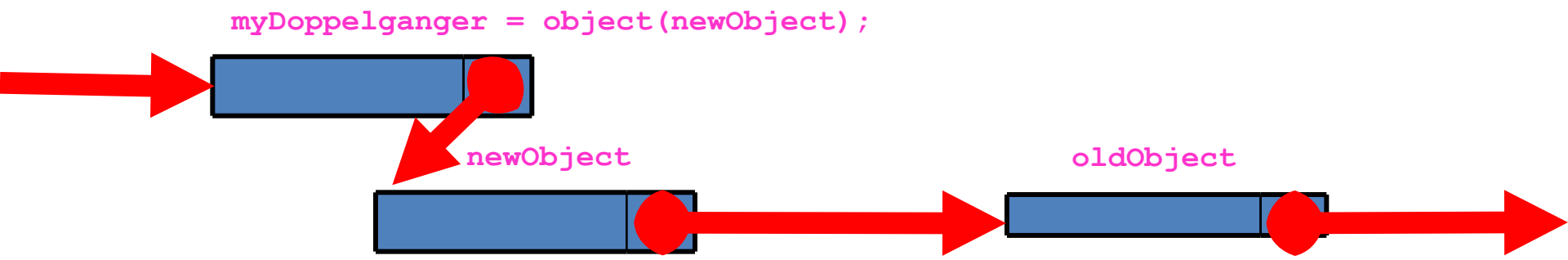


PROTOTYPAL INHERITANCE

```
var oldObject = {
    firstMethod: function () {...},
    secondMethod: function () {...}
};

var newObject = object(oldObject);

newObject.thirdMethod = function () {...};

var myDoppelganger = object(newObject);

myDoppelganger.firstMethod();
```

PROTOTYPAL INHERITANCE

There is no limit to the length of the chain (except common sense).



Now, you can see:

05 JavaScript Introduction to Hierarchy

http://youtu.be/vvld4uzgalw

AUGMENTATION

Using the `object` function, we can quickly produce new objects that have the same state and behavior as existing objects.

We can then augment each of the instances by assigning new methods and members.

PUBLIC METHODS

A Public Method is a function that uses `this` to access its object.

This binding of `this` to an object happens at invocation time.

A Public Method can be reused with many "classes".

PUBLIC METHODS

We can put this function in any object at it works.

```
myObject.method = function (string) {
    return this.member + string;
};
```

Public methods work extremely well with prototypal inheritance and with pseudoclassical inheritance.

SINGLETONS

There is no need to produce a class-like constructor for an object that will have exactly one instance.

Instead, simply use an object literal.

```
var singleton = {
    firstMethod: function (a, b) {
        ...
    },
    secondMethod: function (c) {
        ...
    }
};
```

FUNCTIONS ARE USED AS

Functions

Methods

Constructors

Classes

Modules

MODULE

Variables defined in a module are only visible in the module.

Functions have scope.

Variables defined in a function only visible in the function.

Functions can be used a module containers.

SCOPE

JAVASCRIPT HAS NO BLOCK SCOPE

```
function () {
          var something = true,
         isWrong = true;

     if(something === isWrong) {
       var newVariable = "My ASS!"
     }
     console.log(newVariable); // my ASS!
}
```

JAVASCRIPT HAS FUNCTION SCOPE

```
var something = 'I am Global';

function doSomething() {
          var something = 'I am Local';
     console.log(something); // I am Local
}

doSomething();
console.log(something); // I am Global
```

Functions do not all have to be defined at the top level (or left edge).

INNER FUNCTIONS

Functions can be defined inside of other functions.

SCOPE

An inner function has access to the variables and parameters of functions that it is contained within.

This is known as Static Scoping or Lexical Scoping.

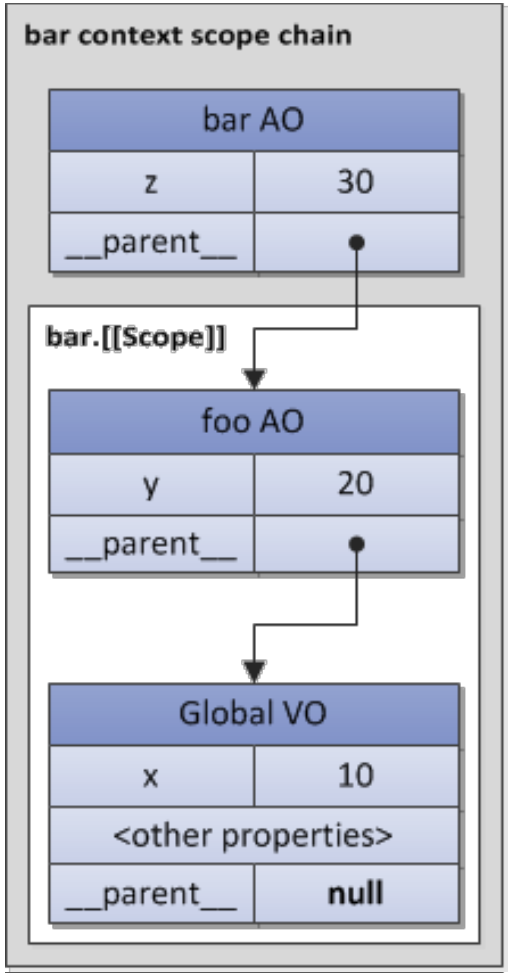
The scope that an inner function enjoys continues even after the parent functions have returned.

This is called *closure*.

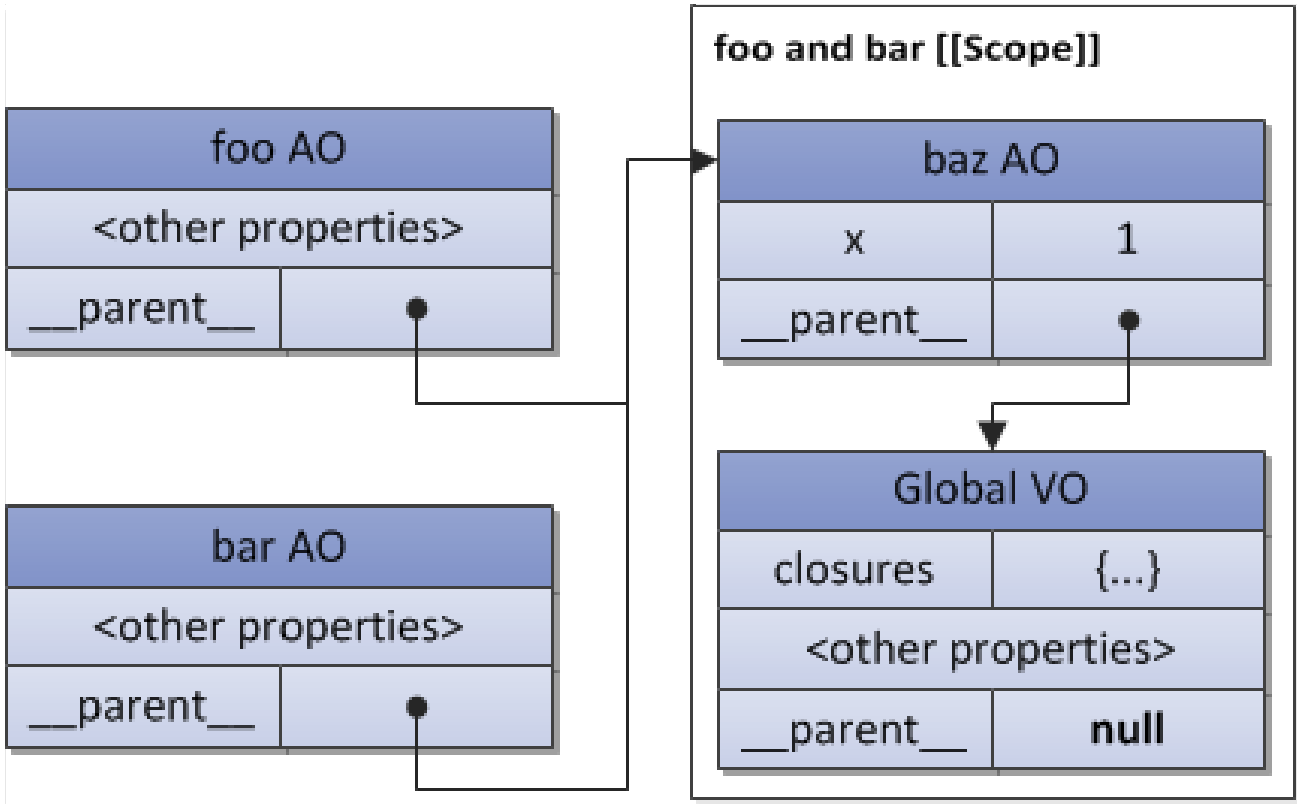
CLOSURE

SCOPE CHAIN

```
var x = 10;

(function foo() {
  var y = 20;
  (function bar() {
    var z = 30;
    // "x" and "y" are "free variables"
    // and are found in the next (after
    // bar's activation object) object
    // of the bar's scope chain
    console.log(x + y + z);
  })();
})();
```



CLOSURE


foo and bar [[Scope]]
foo AO
<other properties>
__parent__
baz AO
x
1
__parent__
bar AO
<other properties>
__parent__
Global VO
closures
{...}
<other properties>
__parent__
null

Now, you can see:

06 Fundamentals of JavaScript

http://youtu.be/1XwHFwGxfI0

CURRYING

Currying is a useful technique, with which you can *partially evaluate* functions.

```
alert(myFunc(2,2));          // alerts 4
var adds4 = myFunc(4); // adds4, is now a function,
                             // which adds 4, to it's argument.
alert(adds4(5));             // alerts 9.
```

It's the second line, which is the key. If you give a curried function, less arguments, then it expects, it will give you back, a function, which has been fed the arguments you gave it, and will accept the remaining ones.

GLOBAL VARIABLES ARE EVIL

Functions within an application can clobber each other.

Cooperating applications can clobber each other.

Use of the global namespace must be minimized.

EXECUTION CONTEXT

However, inside the JavaScript interpreter, every call to an execution context has 2 stages:

1. **Creation Stage** [when the function is called, but before it executes any code inside]:
   - Create variables, functions and arguments.
   - Create the **Scope Chain**.
   - Determine the value of **this**.

2. **Activation / Code Execution Stage**:
   - Assign values, references to functions and interpret / execute code.

EXECUTION CONTEXT

It is possible to represent each execution context conceptually as an object with 3 properties:

```
executionContextObj = {
variableObject: { /* function arguments / parameters,
                   inner variable and function declarations */ },
scopeChain: { /* variableObject + all parent execution context's variableObject */ },
this: {}
}
```

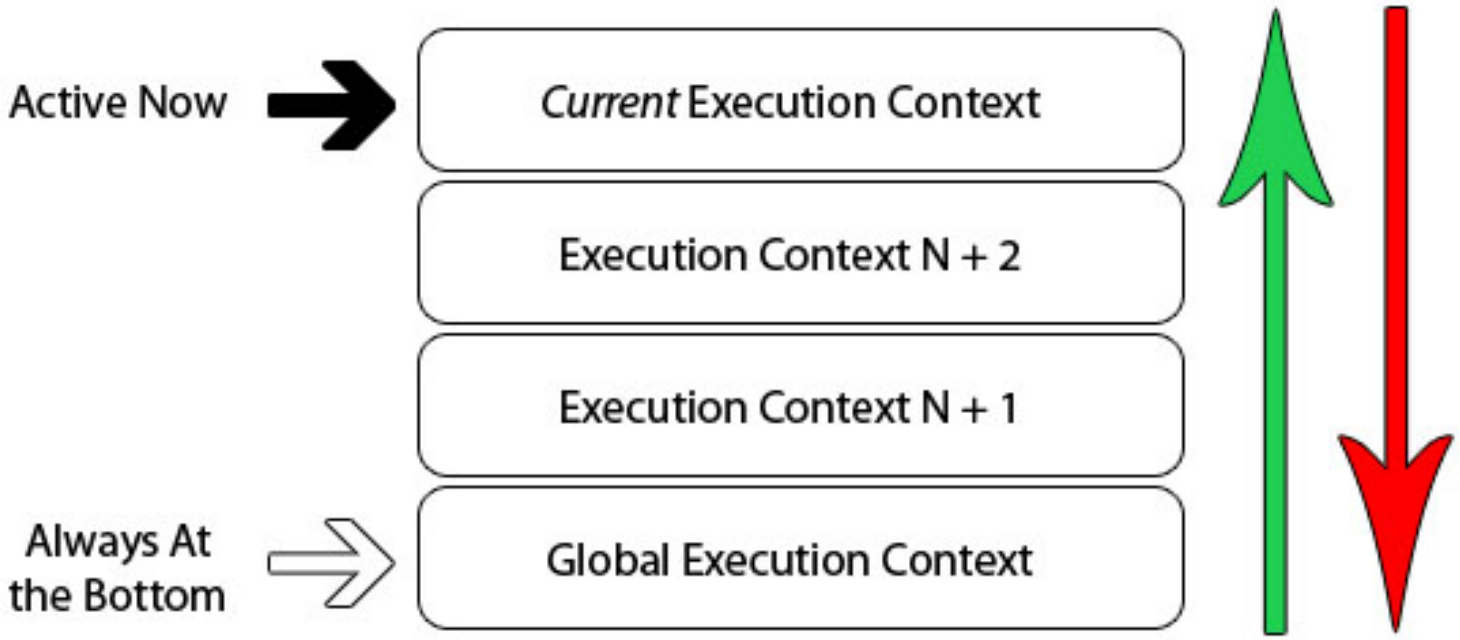
EXECUTION CONTEXT

Execution context	
Variable object	{ vars, function declarations, arguments... }
Scope chain	[Variable object + all parent scopes]
thisValue	Context object

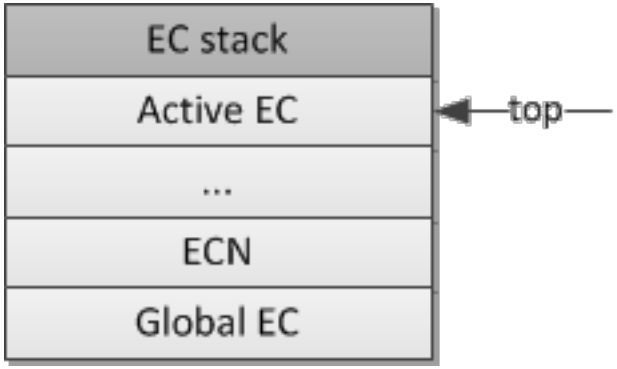
EXECUTION CONTEXT

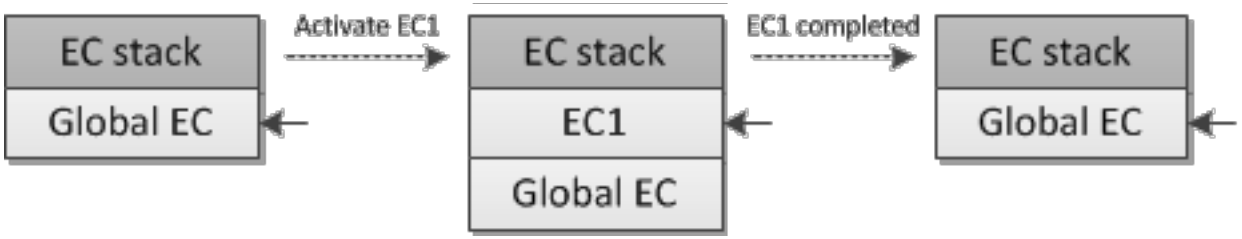
```
// global context

var sayHello = 'Hello';

function person() {          // execution context

    var first = 'David',
        last  = 'Shariff';

    function firstName() {   // execution context
        return first;
    }

    function lastName() {    // execution context
        return last;
    }

    alert(sayHello + firstName() + ' ' + lastName());

}
```

EXECUTION CONTEXT & EC STACK



EXECUTION CONTEXT & EC STACK





Now, you can see:

07 Speed Up Javascript Scope Management

http://youtu.be/tNaIR6Du7vc

The methods of a singleton can enjoy access to shared private data and private methods.

SINGLETONS

SINGLETONS

```
var singleton = function () {
    var privateVariable;
    function privateFunction(x) {
        ...privateVariable...
    }

    return {
        firstMethod: function (a, b) {
            ...privateVariable...
        },
        secondMethod: function (c) {
            ...privateFunction()...
        }
    };
}();
```

APPLICATIONS ARE SINGLETONS

```
var AJAX = = function () {
    var privateVariable;
    function privateFunction(x) {
        ...privateVariable...
    }

    return {
        firstMethod: function (a, b) {
            ...privateVariable...
        },
        secondMethod: function (c) {
            ...privateFunction()...
        }
    };
}();
```

All members of an object are public.

We want private variables and private methods

PRIVACY

PRIVILEGED METHOD

A Privileged Method is a function that has access to secret information.

A Privileged Method has access to private variables and private methods.

A Privileged Method obtains its secret information through closure.

Put the singleton module pattern in constructor function, and we have a power constructor pattern.

POWER CONSTRUCTOR

1. Make a new object somehow.
2. Augment it.
3. Return it.

POWER CONSTRUCTOR

```
function powerConstructor() {
    var that = object(oldObject),
        privateVariable;
    function privateFunction(x) { ... }

    that.firstMethod = function (a, b) {
        ...privateVariable...
    };
    that.secondMethod = function (c) {
        ...privateFunction()...
    };
    return that;
}
```

POWER CONSTRUCTOR

Public methods (from the prototype)

Private variables (var)

Private methods (inner functions)

Privileged methods

No need to use **new**

```
myObject = power_constructor();
```

A power constructor calls another constructor, takes the result, augments it, and returns it as though it did all the work.

PARASITIC INHERITANCE

PSEUDOCLASSICAL INHERITANCE

```
function Gizmo(id) {
    this.id = id;
}
Gizmo.prototype.toString = function () {
    return "gizmo " + this.id;
};

function Hoozit(id) {
    this.id = id;
}
Hoozit.prototype = new Gizmo();
Hoozit.prototype.test = function (id) {
    return this.id === id;
}
```

PARASITIC INHERITANCE

```
function gizmo(id) {
    return {
        id: id,
        toString: function () {
            return "gizmo " + this.id;
        }
    };
}

function hoozit(id) {
    var that = gizmo(id);
    that.test = function (testid) {
        return testid === this.id;
    };
    return that;
}
```

SECRETS

```
function gizmo(id) {
    return {
        toString: function () {
            return "gizmo " + id;
        }
    };
}

function hoozit(id) {
    var that = gizmo(id);
    that.test = function (testid) {
        return testid === id;
    };
    return that;
}
```

SHARED SECRETS

```
function gizmo(id, secret) {
    secret = secret || {};
    secret.id = id;
    return {
        toString: function () {
            return "gizmo " + secret.id;
        };
    };
}

function hoozit(id) {
    var secret = {},
        that = gizmo(id, secret);
    that.test = function (testid) {
        return testid === secret.id;
    };
    return that;
}
```

SUPER METHODS

```
function hoozit(id) {
    var secret = {},
        that = gizmo(id, secret),
        super_toString = that.toString;
    that.test = function (testid) {
        return testid === secret.id;
    };
    that.toString = function () {
        return super_toString.apply(that, []);
    };
    return that;
}
```

EXAMPLE

```
MYAPP.utilities.array = (function () {
  // private properties
  var array_string = "[object Array]",
  ops = Object.prototype.toString,
  // private methods
  inArray = function (haystack, needle) {
    for (var i = 0, max = haystack.length; i < max; i += 1) {
      if (haystack[i] === needle) {
        return i;
      }
    }
    return -1;
  },

  isArray = function (a) {
    return ops.call(a) === array_string;
  };

  // end var
  // revealing public API
  return {
    isArray: isArray,
    indexOf: inArray
  };
}());
```

INHERITANCE PATTERNS

Prototypal Inheritance works really well with public methods.

Parasitic Inheritance works really well with privileged and private and public methods.

Pseudoclassical Inheritance for elderly programmers who are old and set in their ways.

WORKING WITH THE GRAIN

Pseudoclassical patterns are less effective than prototypal patterns or parasitic patterns.

Formal classes are not needed for reuse or extension.

Be shallow. Deep hierarchies are not effective.

Now, you can see:

08 Parasitic Inheritance and Overriding Members

http://youtu.be/WGAwb0bdlh0

Thank you!

Globant
we are ready