JavaScript

Sergio Foti

08/05/2009

JavaScript and OOP

- JavaScript is not a full OOP language, such as Java, but it is an object-based language.

- So, why should you use objects in JavaScript?

  - Not only do they help you better understand how JavaScript works

  - Use it in large scripts, in this way you can create self-contained JavaScript objects, rather than the procedural code you may be using now.

  - Using it also allows you to reuse code more often.

JavaScript is not Java

- What's in a name?
  - In the case of Java and JavaScript, a lot of marketing and relatively little substance.
  - JavaScript was renamed from "livescript" at the last minute by Netscape's marketing department.
- Contrary to popular perception, JavaScript is not a descendent of Java language.
- It has quite a lot in common with functional languages such as Scheme, Self and Python.
- Unfortunately, it's been named Java and syntactically styled to look like Java. In some places it will works like Java, but in many places it simply won't.

About *typeof operator*

- A less-known operator in JavaScript is the ***typeof*** operator.
- It tells you what **type of** data you're handling with.
- Let's look at some examples:

```
var BooleanValue = true;
alert(typeof BooleanValue) // displays "boolean“

var NumericalValue = 354;
alert(typeof NumericalValue) // displays "number"

var StringValue = "This is a String";
alert(typeof StringValue) // displays "string"
```

JavaScript Objects

- Objects are composite data-types.
- They aggregate multiple values into a single unit.
- Allow you to store and retrieve values by name.
- Basically an object is an unordered collection of properties with name value pairs.
- The named values held by an object may be primitive values, such as numbers and strings, functions or they may themselves be objects.

Object Categories

There are three object categories in JavaScript:

- **Native objects** are those objects supplied by JavaScript. Examples of these are String, Number, Array, Image, Date, Math, etc.

- **Host objects** are objects that are supplied to JavaScript by the browser environment. Examples of these are window, document, forms, etc.

- And, **user-defined objects** are those that are defined by you, the programmer.

Literal Objects

- The easiest way to create an object is to include an object literal in your JavaScript code.
- An object literal is a comma-separated list of property name/value pairs, enclosed within curly braces.
- Each property name can be a JavaScript identifier or a string.
- Each property value can be a constant or any JavaScript expression.
- An object literal is an expression that creates and initializes a new and distinct object each time it is evaluated.

Literal Objects

```
// An object with no properties
var empty = {};


// An object with properties
var point = { x:0, y:0 };


// An object with calculated properties
var circle = { x:point.x, y:point.y+1, radius:2 };

// A complex object
var homer = {    "name": "Homer Simpson",
                 "age": 34,
                 "married": true,
                 "occupation": "plant operator",
                 'email': "homer@example.com"
};
```

The new Operator

- The new operator creates a new object and invokes a constructor function to initialize it.

- It is an operator that appears before a constructor invocation, with the following syntax: **new *constructor*(*arguments*)**

  constructor must be an expression that evaluates to a constructor function, and it should be followed by zero or more comma-separated arguments enclosed in parentheses.

- The simplified ***new*** operator flow is:

  - Creates a new object with no properties.

  - Invokes the specified constructor function, passing the specified arguments and also passing the newly created object as the value of the ***this*** keyword.

  - The constructor function can then use the ***this*** keyword to initialize the new object in any way desired.

Native objects

- JavaScript comes with many built-in objects, such as the, Image, and Date objects.

- Many of you are familiar with Image objects from creating rollover effects.

```
// Create a new Image object with no properties
var Image1 = new Image();

// assigne the src property
Image1.src = "myDog.gif";
```

- Using JavaScript's dot-structure (.), in the code above you can accesses and sets the *src* property of your new Image object.

Create user-defined Objects

- But how we can create our own objects?

```
function constructorFunc(){
}
var myFirstObject = new constructorFunc();
alert(typeof myObject);  // displays "object"
```

We've just created our first object!!

- In fact we've created an instance of ***constructorFunc*** object and, as you can see, using ***typeof*** the result will be “object”.

- ***constructorFunc()*** is a constructor function.

- It lays out the blueprint from which objects will be created.

Create user-defined Objects

- How does JavaScript know to create an instance of the ***myFunc*** object, rather than to return its results?
- Let's compare previous example with the more conventional use of a function:

```
function myFunc(){
     return 5;
}
var myFirstObject = myFunc();
alert(typeof myObject);  // displays "number"
```

- In this case, we've assigned 5 to *myFirstObject*.
- What's the difference between these two scripts?

The answer is: the *new* keyword.

- It tells JavaScript to create an object following the blueprint specified in the *myFunc()* constructor function.

Create user-defined Objects

- We've learned how to create a constructor function, in our example, we've created a *myFunc()* constructor function.

- We've learned how to create an object from that constructor function, in our example we've assigned a new instance of *myFunc() to* the variable *myFirstObject*.

Fine but what's the point?

Object properties

- Like a Native object also in user-defined you can assign properties.

```
function myFunc(){
}
var myFirstObject = new myFunc();
myFirstObject.strValue = "This is a String";
alert(myFirstObject.strValue); // displays "This is a String"
```

- We have now created a property and attached dynamically to our object.

Dynamic object properties

- But what's up if we create another instance of the *myFunc* object (using the *myFunc()* constructor function), without assigning it the *strValue* property?

```
function myFunc(){
}
var myFirstObject = new myFunc();
myFirstObject.strValue = "This is a String";
alert(myFirstObject.strValue); // displays "This is a String“
var mySecondObject = new myFunc();
alert(mySecondObject.strValue); // displays undefined
```

- The *strValue* property is assigned only to *myFirstObject* instance but is not defined for all instance of *myFunc* object. Because in this way properties are added dynamically to the object.

Object properties and constructor

- How can we create properties that exist for all *myFunc* object instances?

- The response is: **inside the *myFunc()* constructor function,** that do just that.

```
function myFunc(){
    this.strValue = "This is a String";
}
var myFirstObject = new myFunc();
alert(myFirstObject.strValue); // displays "This is a String“
var mySecondObject = new myFunc();
alert(mySecondObject.strValue); // displays "This is a String“
```

- The ***this*** keyword inside a constructor function refers to the object that's being created.

Object properties and this keyword

- Using ***this*** keyword all *myFunc* object instances will have a *strValue* property.
- The initial value will be assigned to "***This is a String***".
- Every object can have its own distinctive value for the property without affecting the others.

```
function myFunc(){
    this.strValue = “This is a String”;
}
var myFirstObject = new myFunc();
myFirstObject.strValue = “MyFirstObject String”
var mySecondObject = new myFunc();
alert(myFirstObject.strValue);  // displays “MyFirstObject String”
alert(mySecondObject.strValue); // displays “This is a String”
```

Object properties as constructor parameters

- We can valorize an object property passing arguments to our constructor function.

```
function myFunc(strValue){
    this.strValue = strValue;
}
var myFirstObject = new myFunc(“MyFirstObject String”);
var mySecondObject = new myFunc(“This is a String”);
alert(myFirstObject.strValue);  // displays “MyFirstObject String”
alert(mySecondObject.strValue); // displays “This is a String”
```

- In the *myFunc()* constructor, *this.strValue* refers to the property being assigned to the newly created object.
- *strValue* refers to the function's local variable that was passed as an argument.
- So, now we've assigned properties to objects.

JavaScript Variables

- Before you use a variable in JavaScript, you must declare it using the ***var*** keyword, but if you don't declare a variable explicitly, JavaScript will declare it implicitly for you.
- A variable is a name associated with a value, so we can say that the variable stores or contains the value.
- An important difference between JavaScript and languages such as Java and C is that JavaScript is untyped.
- This means that a JavaScript variable can hold a value of any type so perfectly legal to assign a number to a variable and then later assign a string to that variable.
- JavaScript automatically converts values from one type to another, as necessary. i.e. condition evaluation or concatenation.
- If you don't specify an initial value for a variable the variable is declared, but its initial value is ***undefined*** until your code stores a value into it.

Undefined Vs Unassigned Variable

- In JavaScript there are two different kinds of undefined variables:
  - The first kind of undefined variable is one that has never been declared, *undeclared* variable.
    - An attempt to read the value of an undeclared variable causes a runtime error, because they simply do not exist.
    - Assigning a value to an undeclared variable does not cause an error; instead, it implicitly declares the variable in the global scope.
  - The second kind of undefined variable is one that has been declared but has never had a value assigned to it, *unassigned* variable.
    - If you read the value of one of these variables, you obtain its default value, ***undefined***.

The importance of Variable scope

- The scope of a variable is the region of your program in which it is defined.

- There are two kind of variable scope:
  - Variables with **global scope**; they are accessible everywhere in your JavaScript code.
  - Variables with **local scope**: declared within a function are defined only within the body of the function. Function parameters also count as local variables and are defined only within the body of the function.

- If you declare a local variable or function parameter with the same name as a global variable, you effectively hide the global variable.

- Note that within the body of a function, a local variable takes precedence over a global variable with the same name. To avoid this problem **declare all variables with *var* keyword**.

The importance of Variable scope

- Just some examples

```
var scope = “global”;
function myFunc(){
     scope = “Modified value”;
     alert(scope);
}
myFunc(); // displays “Modified value” String
alert(scope);    // displays “Modified value” String
```

Remember to use ***var*** keyword to avoid mistake

- The right way (variable hiding)

```
var scope = “global”;
function myFunc(){
     var scope = “Modified value”;
     alert(scope);
}
myFunc(); // displays “Modified value” String
alert(scope);    // displays “global” String
```

Variable - No Block Scope

- Note that unlike C, C++, and Java, JavaScript does not have block-level scope.
- All variables declared in a function, no matter where they are declared, are defined throughout the function.
- In the following code, the variables i, j, and k all have the same scope: all three are defined throughout the body of the function.

```
function myFunc(obj){
    var i = 0; // i is defined throughout function
    if (typeof obj == "object") {
        var j = 0; // j is defined everywhere, not just block
        for(var k=0; k < 10; k++) {
            // k is defined everywhere, not just loop
            document.write(k);
        }
        document.write(k); // k is still defined: prints 10
    }
    document.write(j); // j is still defined: prints 0
}
```

Variable - No Block Scope

- The rule that all variables declared in a function are defined throughout the function can cause surprising results.

```
var scope = "global";
function myFunc(){
    alert(scope); // Displays "undefined", not "global"
    // scope is initialized here, but defined everywhere
    var scope = "local";
    alert(scope); // Displays "local"
}
```

- You might think that the first call to ***alert( )*** would display ***"global"***, because the ***var*** statement, declaring the local variable, has not yet been executed.

- **False**, because local variable is defined throughout the body of the function, which means the global variable by the same name is hidden throughout the function.

- The local variable is defined but not initialized until the ***var*** statement is executed.

Variable - No Block Scope

- The rule that all variables declared in a function are defined throughout the function can cause surprising results.

```
var scope = "global";
function myFunc(){
     alert(scope); // Displays "undefined", not "global"
     // scope is initialized here, but defined everywhere
     var scope = scope ;
     alert(scope); // Displays "undefined", not "global"
}
```

- The explanation of the behavior (*undefined, undefined*) is that the line

  `var scope = scope;`

  is doing at the same time :
  - declaration ***`var x;`***
  - an initialization ***`scope = scope;`***

  But each of these processes happen separately, one after the other, when the javascript code is interpreted and then executed.
- First, when the code is interpreted, before any code is executed, the var x; declaration makes the variable x available within the scope of the entire function, no matter what line it is written on.
- However, the variable won't be initialized, i.e., assigned to any value, until the code is executed. Then, its default value will be ***undefined***.
- Second, when the code is executed, any x inside that function will refer to the declared local x instead of the global x as one may think at first sight. Therefore, the x = x; initialization won't throw any run time error because the variable x exists but it has never been assigned to any value other than the default undefined.

Variable - No Block Scope

- Another common error may occurs during variable declaration:

```
function myFunc(){
    var a = b = 0;
}
```

- Only the variable *a is* declared in the scope of the function.

- Instead variable *b* is not declared so the variable will be created in the global scope and initialized to 0.

```
function myFunc(){
    var a = 0, b = 0;
}
```

- Now, the ***var*** operator applies to both *a* and *b*, creating two local variables (the comma indicates another variable being defined).

Functions

- A function is a block of JavaScript code that is defined once but may be invoked, or executed, any number of times.
- Functions may have parameters (arguments) local variables whose value is specified when the function is invoked.
- Functions often use these arguments to compute a return value that becomes the value of the function-invocation expression.
- When a function is invoked on an object, the function is called as method, and the object on which it is invoked is passed as an implicit argument of the function (this).

Defining Functions

- The most common way to define a function is with the ***function*** statement.
- This statement consists of the function keyword followed by:
  - The name of the function
  - Zero or more parameter names contained within parenthesis, each separated by commas
  - The JavaScript statements that comprise the body of the function, contained within curly braces

Functions return statement

- Function may be defined to expect varying numbers of *arguments* and that they may or may not contain a *return* statement.

  - The *return* statement causes the function to stop executing and to return the value of its expression (if any) to the caller.

  - If the *return* statement does not have an associated expression, it returns the ***undefined*** value.

  - If a function does not contain a *return* statement, it simply executes each statement in the function body and returns the ***undefined*** value to the caller.

  - JavaScript's automatic semicolon insertion, **you may not include a line break between the return keyword and the expression that follows it**.

Function arguments

- JavaScript is a loosely typed language, so you are not expected to specify a data type for function parameters.
  - If the data type of an argument is important, you can test it yourself with the ***typeof*** operator.

- JavaScript does not check whether you have passed the correct number of arguments either.
  - If you pass more arguments than the function expects, the function ignores the extra arguments.
  - If you pass fewer than expected, the parameters you omit are given the ***undefined*** value.
  - Functions can be written to tolerate omitted arguments, or can checks if you don't pass all the arguments they expect.

Optional Arguments

- When a function is invoked with fewer arguments than are declared, the additional arguments have the ***undefined*** value.
- It is often useful to write functions so that some arguments are optional and may be omitted when the function is invoked. **Remember that to do this you must assign a reasonable default value to omitted arguments**.

```
function copyPropertyNamesTOArray(o, /* optional */ a) {
    if (!a) a = []; // If undefined or null, use a blank array
    for(var key in o)
        a.push(o[key]);
    return a;
}
```

Optional Arguments

- With the function defined this way, you have flexibility in how it is invoked:

```
// Get a1 properties into a new array
var a = copyPropertyNamesToArray(a1);
// append a1 properties to that array a2
copyPropertyNamesToArray(a1, a2);
```

- Instead of using an ***if*** statement in the first line of this function, you can use the || operator in this way:

    a = a || [];

- The **||** operator returns its first argument if that argument is ***true*** (or a value that converts to ***true***). Otherwise it returns its second argument.
- In our case, it returns ***a2***, if ***a2*** is defined and non-null, even if it is empty. Otherwise, it returns a new, empty array.

Variable Argument list

- Within the body of a function, the identifier ***arguments*** has special meaning.
- It is an array-like object that allows you to retrieve by number the argument values passed to the function, rather than by name.
- Although a JavaScript function is defined with a fixed number of named arguments, it can be invoked with any number of arguments.
- The ***arguments*** object allows full access to these argument values, even when some or all are unnamed.

```
function myFunc(param1){
    alert(param1);         // Display the String "value1"
    alert(arguments[1]); // Display the String "value2"
}
myFunc("value1", "value2");
```

- ***myFunc*** expects only one argument, ***param1***.
- If you invoke this function with two arguments, the first argument is accessible by the parameter name *param1*.
- The second argument is accessible only as ***arguments[1]***.

Variable Argument list length

- Like true arrays, ***arguments*** has a ***length*** property that specifies the number of elements it contains.

```
function myFunc(x, y, z) {
    // Verify the right number of passed arguments
    if(arguments.length != 3) {
        throw new Error(“Error...“);
    }
    // Now do the actual function...
}
```

- The previous example shows how you can use it to verify that a function is invoked with the correct number of arguments, since JavaScript doesn't do this for you.

- Note that arguments is an ordinary JavaScript identifier, not a reserved word. If a function has an argument or local variable with that name, it hides the reference to the Arguments object. For this reason, it is a good idea to **treat arguments as a reserved word and avoid using it as a variable name**.

Argument *callee* property

- Arguments object defines a ***callee*** property that refers to the function that is currently being executed.

- This property is rarely useful, but it can be used to allow unnamed functions to invoke themselves recursively.

```
function(x){
    if (x <= 1)
        return 1;
    return x * arguments.callee(x-1);
}
```

Using Object as Arguments

- When a function requires more than about three arguments, it becomes difficult for the programmer who invokes the function to remember the correct order in which to pass arguments.

- Define your function to expect a single object as its argument and then users have to pass pass an object literal that defines the required name/value pairs.

```
function arraycopy(from, from_start, to, to_start, length) {
    // code goes here
}
```

- It is hard to remember the order of the arguments.

```
function easycopy(args){
    // here you can use args.from to access parameters
}
easycopy({from: [1,2,3], to: new Array(3), length: 3});
```

- A little less efficient, but you don't have to remember the order of the arguments.

Nested Functions

- In JavaScript, functions may be nested within other functions.

```
function hypotenuse(a, b){
    function square(x) {
        return x*x;
    }
    return Math.sqrt(square(a) + square(b));
}
```

- Nested functions may be defined only at the top level of the function within which they are nested. They may not be defined within statement blocks, such as the body of an if statement or while loop.

- Note that this restriction applies only to functions defined with the ***function*** statement. Function literal expressions may appear anywhere.

Function Literals

- A function literal is defined with the *function* keyword, followed by an optional function name, followed by a parenthesized list of function arguments, and then the body of the function within curly braces.
- A function literal looks just like a function definition, except that it does not have to have a name.
- The big difference is that function literals can appear within other JavaScript expressions.

```
function square(x) { return x*x; }

it can be defined with a function literal:

var square = function(x) { return x*x; }
```

- Although it is not immediately obvious why you might choose to use function literals in a program, but as we will see later it can be used as object methods.

Functions as Data

- In JavaScript, functions are not only syntax but also data. It can be assigned to variables, stored in the properties of objects or the elements of arrays, but also **passed as arguments to other functions**.

```
// We define some simple functions here
function add(x,y) { return x + y; }
function subtract(x,y) { return x - y; }
function multiply(x,y) { return x * y; }
function divide(x,y) { return x / y; }

// Here's a function that takes one of the above functions
// as an argument and invokes it on two operands
function operate(operator, operand1, operand2) {
    return operator(operand1, operand2);
}
// We could invoke this function like this
var i = operate(add, operate(add, 2, 3), operate(multiply, 4, 5));
```

Functions as Methods

- A method is nothing more than a JavaScript function that is stored in a property of an object and invoked through that object, it can be assigned to any variable, or even to any property of an object.

- If you have a function ***fnc*** and an object ***obj***, you can define a method named ***method*** with the following line:

  ```
  obj.method = fnc;
  ```

  and invoke it like this: ***`obj.method()`***;

- Methods have one very important property, the object through which a method is invoked becomes the value of the ***this*** keyword within the body of the method. This mean that in the body of the method you can refer he object with the ***this*** keyword.

- Any function that is used as a method have effectively passed an implicit argument that is the object through which it is invoked.

Functions as Methods

- Typically, a method performs some operation on that object, so the method-invocation syntax is an elegant way to express the fact that a function is operating on an object.

- Compare the following two lines of code:

  `rect.setSize(width, height); setRectSize(rect, width, height);`

- The functions invoked in these two lines of code may perform the same operation on the object ***rect***, but the method-invocation syntax in the first line more clearly indicates the idea that it is the object ***rect*** that is the primary focus of the operation.

- When a function is invoked as a function rather that as a method, the ***this*** keyword refers to the global object.

- Note that ***this*** is a keyword, not a variable or property name. JavaScript syntax does not allow you to assign a value to ***this***.

The apply() and call() Methods

- ECMAScript specifies two methods that are defined for all functions, ***call()*** and ***apply()***.
- These methods allow you to invoke a function as if it were a method of some other object.
- The first argument to both is the object that becomes the value of the ***this*** keyword within the body of the function.
- Remaining arguments for ***call()*** method are the values that will be passed to the function that is invoked.
- The ***apply()*** method is like the ***call()*** method, but the arguments that will be passed to the function are specified as an array.

```
// We define some simple functions here
function add(x,y) { return x + y; }

add.call(obj, 1, 2);

add.apply(obj, [1, 2]);
```

Function – Lexical Scope

- Functions in JavaScript are lexically scoped.
- This means that they run in the scope in which they are defined, not the scope from which they are executed.
- When a function is defined, the current scope chain is saved and becomes part of the internal state of the function.
- At the top level, the scope chain simply consists of the global object, and lexical scoping is not particularly relevant.
- When you define a nested function, however, the scope chain includes the containing function. This means that nested functions can access all of the arguments and local variables of the containing function.
- **Note:** although the scope chain is fixed when a function is defined, the properties defined in that scope chain are not fixed. The scope chain is "live," and a function has access to whatever bindings are current when it is invoked.

Function – The Call Object

- When the JavaScript interpreter invokes a function, it first sets the scope to the scope chain that was in effect when the function was defined.
- Next, it adds a new object, known as the ***call object*** (the ECMAScript specification uses the term *activation object*) to the front of the scope chain.
- Then the call object is initialized with a property named arguments that refers to the Arguments object for the function.
- Next named parameters of the function are added to the call object.
- Any local variables declared with the var statement are also defined within this object.

NOTE: Since this call object is at the head of the scope chain, local variables, function parameters, and the Arguments object are all in scope within the function. This also means that they hide any properties with the same name that are further up the scope chain, of course. Moreover take care that unlike arguments, ***this*** is a keyword and not a property in the call object.

The Call Object as a Namespace

- It is sometimes useful to define a function simply to create a call object that acts as a temporary namespace in which you can define variables and create properties without corrupting the global namespace.

- Suppose, for example, you have a file of JavaScript code that you want to use in a number of different JavaScript programs (or, for client-side JavaScript, on a number of different web pages).

- Assume that this code, like most code, defines variables to store the intermediate results of its computation. The problem is that since this code will be used in many different programs, you don't know whether the variables it creates will conflict with variables used by the programs that import it.

The Call Object as a Namespace

- The solution, of course, is to put the code into a function and then invoke the function. This way, variables are defined in the call object of the function.

```
function init() {
    // Code goes here.
    // Any variables declared become properties of the call
    // object instead of cluttering up the global namespace.
}
init(); // But don't forget to invoke the function!
```

- The code adds only a single property to the global namespace: the property "init", which refers to the function.

Anonymous function

- If defining even a single property is too much, you can define and invoke an anonymous function in a single expression.
- The code for this JavaScript idiom looks like this:

```
(function {
    // Code goes here.
    // Any variables declared become properties of the call
    // object instead of cluttering up the global namespace.
})(); // end the function literal and invoke it now.
```

Note: the parentheses around the function literal are required by JavaScript syntax.

Nested Functions as Closures

- The facts that JavaScript allows nested functions, allows functions to be used as data, and uses lexical scoping interact to create surprising and powerful effects.

```
function f() {
     function g() { alert(x); }
    g();
}
f();
```

- Consider a function ***g*** defined within a function ***f***.
- When ***f*** is invoked, the scope chain consists of the *call object* for that invocation of ***f*** followed by the *global object*.
- ***g*** is defined within ***f***, so this scope chain is saved as part of the definition of ***g***.
- When ***g*** is invoked, the scope chain includes three objects: its own *call object*, the *call object* of ***f***, and the *global object*.

The Function() Constructor

- Functions are usually defined using the ***function*** keyword, but can also be defined with the ***Function()*** constructor.

- Using the ***Function()*** constructor is typically harder than using a function literal, and so this technique is not commonly used, but you can find it in some scripts.

```
// Here a literal function
var f = function add(x, y) { return x + y; }

// Here a function using Function constructor
var f = new Function("x", "y", "return x + y;")
```

This script are more or less equivalent

- The ***Function()*** constructor expects any number of string arguments, that specify the names of the parameters to the function being defined.

- The last argument is the body of the function; it can contain arbitrary JavaScript statements, separated from each other by semicolons.

- If you are defining a function that takes no arguments, you simply pass a single string that will be the function body.

The Function() Constructor

- The ***Function()*** constructor allows JavaScript code to be **dynamically** created and compiled at runtime. It is like the global ***eval()*** function.
- The ***Function()*** constructor parses the function body and creates a new function object each time it is called.
- By contrast, using a function literal the objects are not created each time ***function*** keyword is encountered.
- If the call to the constructor appears within a loop or within a frequently called function, this process can be inefficient.
- By contrast, a ***function*** literal or nested function that appears within a loop or a function, is not recompiled each time it is encountered.

The Function() Constructor

- The ***Function()*** constructor do not use lexical scoping, they are always compiled as if they were top-level functions.

```
var y = "global";
function constructFunction() {
     var y = "local";
     return new Function("return y");
}
var fnc = constructFunction();
alert(fnc());
```

This line will display "global"

```
var y = "global";
function constructFunction() {
     var y = "local";
     return new Function("return " + y + ";");
}
var fnc = constructFunction();
alert(fnc());
```

Now will display "local"

Type Conversion Summary

- The basic rule is that when a value of one type is used in a context that requires a value of some other type, JavaScript automatically attempts to convert the value as needed.
- So, for example, if a number is used in a Boolean context, it is converted to a boolean.
- If an object is used in a string context, it is converted to a string.
- If a string is used in a numeric context, JavaScript attempts to convert it to a number.

Type Conversion Summary Table

- Following table summarizes each conversions and shows the conversion that is performed when a particular type of value is used in a particular context.

Value	**Context in which value is used**			
	String	**Number**	**Boolean**	**Object**
Undefined value	"undefined"	NaN	FALSE	Error
null	"null"	0	FALSE	Error
Nonempty string	As is	Numeric value of string or NaN	TRUE	String object
Empty string	As is	0	FALSE	String object
0	"0"	As is	FALSE	Number object
NaN	"NaN"	As is	FALSE	Number object
Infinity	"Infinity"	As is	TRUE	Number object
Negative infinity	"-Infinity"	As is	TRUE	Number object
Any other number	String value of number	As is	TRUE	Number object
TRUE	"true"	1	As is	Boolean object
FALSE	"false"	0	As is	Boolean object
Object	toString( )	valueOf( ), toString( ), or NaN	TRUE	As is