JavaScript OOP Concepts

Differ with Other Programming Language:

1. JavaScript is Prototype based object oriented language instead of class based language.
2. It is easy to define new properties and methods in JS at any moment which is difficult in class based language like C++, C# or others.
3. JavaScript has no concepts of Classes like traditional Object Oriented languages.

Some Practices:

1. In comparison use triple equals instead of double equals so that you can correctly identify the type
2. It’s best to dereference objects that you no longer need so that the garbage collector can free up that memory. The best way to do this is to set the object variable to null. Dereferencing objects is especially important in very large applications that use millions of objects. Example –

var object1 = new Object();

// do something

object1 = null; // dereference

1. In practice, checking the named parameter against undefined is more common than relying on arguments.length.

Encapsulation:

* Data can be grouped together with functionality that operates on that data. This, quite simply, is the definition of an object.

Aggregation:

* One object can reference another object

Inheritance:

* A newly created object has the same characteristics as another object without explicitly duplicating its functionality.

Polymorphism:

* One interface may be implemented by multiple objects.

Types:

* Primitive types are stored as simple data types.
* Reference types are stored as objects, which are really just references to locations in memory.
* The tricky thing is that JavaScript lets you treat primitive types like reference types in order to make the language more consistent for the developer.
* While other programming languages distinguish between primitive and reference types by storing primitives on the stack and references in the heap.
* JavaScript does away with this concept completely: It tracks variables for a particular scope with a variable object.
* Primitive values are stored directly on the variable object, while reference values are placed as a pointer in the variable object, which serves as a reference to a location in memory where the object is stored.

Identifying Primitive Types:

* The best way to identify primitive types is with the typeof operator, which works on any variable and returns a string indicating the type of data.

Example of typeof Operator:

* console.log(typeof "Nicholas"); // "string"
* console.log(typeof 10); // "number"
* console.log(typeof 5.1); // "number"
* console.log(typeof true); // "boolean"
* console.log(typeof undefined); // "undefined"

Objects

* An object is an unordered list of properties consisting of a name (always a string) and a value.
* When the value of a property is a function, it is called a method.

New Keyword in Constructor

The convention is that constructor functions should begin with a capital letter. Note: if the new keyword is not used, then the 'this' variable inside the function will refer to the global object. Can you smell a potential mess? Hence why the capital letter convention for constructor functions is used. The capital letter means: "I am a constructor function, please use the new keyword".

Creating Objects

* The first is to use the new operator with a constructor.
* By convention, constructors in JavaScript begin with a capital letter to distinguish them from nonconstructor functions. Like

var object = new Object();

Adding or Removing Properties

* Another interesting aspect of objects in JavaScript is that you can add and remove properties at any time.
* For example:

var object1 = new Object();

var object2 = object1;

object1.myCustomProperty = "Awesome!";

console.log(object2.myCustomProperty); // "Awesome!"

* Here, myCustomProperty is added to object1 with a value of "Awesome!". That property is also accessible on object2 because both object1 and object2 point to the same object.

Instantiating Built-in Types:

* var items = new Array();
* var now = new Date();
* var error = new Error("Something bad happened.");
* var func = new Function("console.log('Hi');");
* var object = new Object();
* var re = new RegExp("\\d+");

Object Literals

* To create an object with object literal syntax, you can define the properties of a new object inside braces

var book = {

name: "The Principles of Object-Oriented JavaScript",

year: 2014

};

* Example of String Literals:

var book = {

"name": "The Principles of Object-Oriented JavaScript",

"year": 2014

};

* Example of String Literals:

var book = {

"name": "The Principles of Object-Oriented JavaScript",

"year": 2014

};

Array Literals

* We can define an array literal in a similar way by enclosing any number of comma-separated values inside square brackets.

For example:

var colors = [ "red", "blue", "green" ];

console.log (colors [0]); // "red"

This code is equivalent to the following:

var colors = new Array("red", "blue", "green")

console.log (colors [0]); // "red"

Summary:

* While JavaScript doesn’t have classes, it does have types. Each variable or piece of data is associated with a specific primitive or reference type. The five primitive types (strings, numbers, Booleans, null, and undefined) represent simple values stored directly in the variable object for a given context.
* We can use typeof to identify primitive types with the exception of null, which must be compared directly against the special value null.
* Reference types are the closest thing to classes in JavaScript, and objects are instances of reference types.
* We can create new objects using the new operator or a reference literal.
* We access properties and methods primarily using dot notation, but you can also use bracket notation.
* Functions are objects in JavaScript, and you can identify them with the typeof operator. You should use instanceof with a constructor to identify objects of any other reference type. To make primitives seem more like references, JavaScript has three primitive wrapper types: String, Number, and Boolean. JavaScript creates these objects behind the scenes so that you can treat primitives like regular objects, but the temporary objects are destroyed as soon as the statement using them is complete. Although you can create your own instances of primitive wrappers, it’s best not to do that because it can be confusing.

Functions

* There are actually two literal forms of functions.
* The first is a function declaration, which begins with the function keyword and includes the name of the function immediately following it.

function add(num1, num2) {

return num1 + num2;

}

* The second form is a function expression, which doesn’t require a name after function. These functions are considered anonymous because the function object itself has no name.

var add = function(num1, num2) {

return num1 + num2;

};

* Functions as Values:

function sayHi() {

console.log("Hi!");

}

* sayHi(); // outputs "Hi!"
* var sayHi2 = sayHi;
* sayHi2(); // outputs "Hi!"

Parameters in Functions:

* We can use named parameter in functions like

function reflect(value) {

return value;

}

* console.log(reflect("Hi!")); // "Hi!"
* console.log(reflect("Hi!", 25)); // "Hi!"
* console.log(reflect.length); // 1

Pass arguments in functions like

function sum() {

var result = 0,

i = 0,

len = arguments.length;

while (i < len) {

result += arguments[i];

i++;

}

return result;

}

* console.log(sum(1, 2)); // 3
* console.log(sum(3, 4, 5, 6)); // 18
* console.log(sum(50)); // 50
* console.log(sum()); // 0
* Sometimes, however, using arguments is actually more effective than naming parameters. For instance, suppose you want to create a function that accepts any number of parameters and returns their sum. You can’t use named parameters because you don’t know how many you will need, so in this case, using arguments is the best option.

Overloading:

* Ability of a single function to have multiple signatures.
* A function signature is made up of the function name plus the number and type of parameters the function expects.
* Thus, a single function can have one signature that accepts a single string argument and another that accepts two numeric arguments. The language determines which version of a function to call based on the arguments that are passed in.

This Objects

* When you say this, you're actually saying "this object" or "the current object". Example:

function sayNameForAll() {

console.log(this.name);

}

var person1 = {

name: "Nicholas",

sayName: sayNameForAll

};

var person2 = {

name: "Greg",

sayName: sayNameForAll

};

var name = "Michael";

* person1.sayName(); // outputs "Nicholas"
* person2.sayName(); // outputs "Greg"
* sayNameForAll(); // outputs "Michael"

Changing This

Three Methods:

* The call() Method
* The apply() Method
* The bind() Method