Apptron Technologies

Objects

An object is an instance which contains a set of key value pairs. Unlike primitive data types, objects can represent multiple or complex values and can change over their life time. The values can be scalar values or functions or even array of other objects

Object Initializers:

Like the primitive types, objects have a literal syntax: curly bracesv ({and}). Following is the syntax for defining an object.

```
var identifier = {

 Key1:value,

 Key2: function () {

   //functions

 },

 Key3: [“content1”,” content2”]

}
```

The contents of an object are called properties (or members), and properties consist of a name (or key) and value

objectName.propertyName

```
var person = {

 firstname:"App",

 lastname:"TRON",
```

```
  func:function(){return "Hello!!"},

};

//access the object values

console.log(person.firstname)

console.log(person.lastname)

console.log(person.func())
```

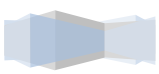
```
var foo = 'bar'

var baz = { foo }

console.log(baz.foo)
```

```
var foo = 'bar'

var baz = { foo:foo }

console.log(baz.foo)
```

The Object() Constructor:

JavaScript provides a special constructor function called Object() to build the object. The new operator is used to create an instance of an object. To create an object, the new operator is followed by the constructor method.

```
var obj_name = new Object();
```

obj_name.property = value;

OR

obj_name["key"] = value

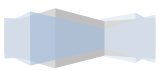
Object_name.property_key

OR

Object_name["property_key"]

Exmples :

```
var myCar = new Object();
myCar.make = "Ford"; //define an object
myCar.model = "Mustang";
myCar.year = 1987;

console.log(myCar["make"]) //access the object property
console.log(myCar["model"])
console.log(myCar["year"])

var myCar = new Object();
myCar.make = "Ford";
console.log(myCar["model"])
```

```
var myCar = new Object()

var propertyName = "make";

myCar[propertyName] = "Ford";

console.log(myCar.make)
```

Constructor Function

An object can be created using the following two steps

Step 1 :

Define the object type by writing a constructor function.

Following is the syntax for the same.

```
function function_name() {
  this.property_name = value
}
```

The ‘this’ keyword refers to the current object in use and defines the object’s property.

Step 2:

Create an instance of the object with the new syntax.


```
var Object_name= new function_name()
```

//Access the property value

Object_name.property_name

The new keyword invokes the function constructor and initializes the function's property keys.

Using a Function Constructor:

```
function Car() {
  this.make = "Ford"
  this.model = "F123"
}
var obj = new Car()
console.log(obj.make)
console.log(obj.model)
```

```
Ford
F123
```

A new property can always be added to a previously defined object. For example,

```
function Car() {
  this.make = "Ford"
}
var obj = new Car()
```

```
obj.model = "F123"

console.log(obj.make)

console.log(obj.model)
```

```
Ford

F123
```

The Object create Method

Objects can also be created using the Object.create() method. It allows you to create the prototype for the object you want, without having to define a constructor function.

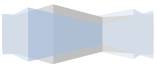
```
var roles = {
  type: "Admin", // Default value of properties
  displayType : function() {
    // Method which will display type of role
    console.log(this.type);
  }
}
// Create new role type called super_role
var super_role = Object.create(roles);
super_role.displayType(); // Output:Admin
```

```
// Create new role type called Guest

var guest_role = Object.create(roles);

guest_role.type = "Guest";

guest_role.displayType(); // Output:Guest


out:

Admin

Guest
```

The Object.assign() Function

The Object.assign() method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object.

```
Object.assign(target, ...sources)


"use strict"

var det = {

        name:"Tom",

        ID:"E1001"

};

var copy = Object.assign({}, det);

console.log(copy);
```

```
for (let val in copy) {
  console.log(copy[val])
}
```

result :

Tom

E1001

Example Merging

```
var o1 = { a: 10 };
var o2 = { b: 20 };
var o3 = { c: 30 };
var obj = Object.assign(o1, o2, o3);
console.log(obj);
console.log(o1);
```

Deleting Properties

```
// Creates a new object, myobj, with two properties, a and b.
var myobj = new Object;
```

```
myobj.a = 5;

myobj.b = 12;


// Removes the ‘a’ property

delete myobj.a;

console.log ("a" in myobj) // yields "false"
```

out put :

```
false


var val1 = {name: "Tom"};

var val2 = val1


console.log(val1 == val2) // return true

console.log(val1 === val2) // return true
```

Map

Map is a data collection type (in a more fancy way abstract data structure type), in which, data is stored in a form of pairs, which contains a unique key and value mapped to that key. And because of the uniqueness of each stored key, there is no duplicate pair stored.

Example: {(1, “smile”), (2, “cry”), (42, “happy”)}

Object:

The ability to create JavaScript objects using literal notation is powerful. New features introduced from ES2015 (ES6)

```
var myObject = {
 prop1: 'hello',
 prop2: 'world',
 output: function() {
  console.log(this.prop1 + ' ' + this.prop2);
 }
};

myObject.output(); // hello world
```

Single-use objects are used extensively. Examples include configuration settings, module definitions, method parameters, return values from functions, etc. ES2015 (ES6) added a range of features to enhance object literals

Object Initialization From Variables

Objects' properties are often created from variables with the same name. For example:

```
var
 a = 1, b = 2, c = 3;
 obj = {
```

```
  a: a,
  b: b,
  c: c
 };

// obj.a = 1, obj.b = 2, obj.c = 3
```

Object Method Definition Shorthand

1) 0bject methods in ES5 require the function statement. For example:

```
// ES5 code
var lib = {
 sum:  function(a, b) { return a + b; },
 mult: function(a, b) { return a * b; }
};

console.log( lib.sum(2, 3) );  // 5
console.log( lib.mult(2, 3) ); // 6
```

2) This is no longer necessary in ES6; it permits the following shorthand syntax:

```
// ES6 code
const lib = {
```

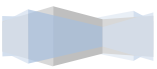
```
 sum(a, b)  { return a + b; },
 mult(a, b) { return a * b; }
};

console.log( lib.sum(2, 3) );  // 5
console.log( lib.mult(2, 3) ); // 6
```

3) It's not possible to use ES6 fat arrow => function syntax here, because the method requires a name. That said, you can use arrow functions if you name each method directly (like ES5). For example:

```
// ES6 code
const lib = {
 sum:  (a, b) => a + b,
 mult: (a, b) => a * b
};

console.log( lib.sum(2, 3) );  // 5
console.log( lib.mult(2, 3) ); // 6
```

Dynamic Property Keys

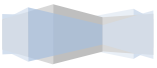
In ES5, it wasn't possible to use a variable for a key name, although it could be added after the object had been created. For example



```
// ES5 code
var
 key1 = 'one',
 obj = {
  two: 2,
  three: 3
 };

obj[key1] = 1;

// obj.one = 1, obj.two = 2, obj.three = 3
```

Object keys can be dynamically assigned in ES6 by placing an expression in [square brackets]. For example:

```
// ES6 code
const key1 = 'one',
 obj = {
  [key1]: 1,
  two: 2,
  three: 3
 };

// obj.one = 1, obj.two = 2, obj.three = 3
```

Any expression can be used to create a key. For example:

```
// ES6 code
const i = 1,
  obj = {
    ['i' + i]: i
  };

console.log(obj.i1); // 1
```

A dynamic key can be used for methods as well as properties. For example:

```
// ES6 code
const
  i = 2,
  obj = {
    ['mult' + i]: x => x * i
  };

console.log( obj.mult2(5) ); // 10
```


Whether you should create dynamic properties and methods is another matter. The code can be difficult to read, and it may be preferable to create object factories or classes.

ADVANCE TOPIC

Property accesses :

properties of a javascript object can be accessed by dot notation or bracket notation as shown below

```
let bike = {
        name: 'SuperSport',
        maker:'Ducati',
        engine:'937cc'
        };
console.log(bike.engine);    //Output: '937cc'
console.log(bike['maker']);  //Output: 'Ducati'
```

Adding property to the object

To add property to the already created object, no need to change the existing object literal, property can be added later with dot notation as shown below

```
let bike = {
        name: 'SuperSport',
```

```
            maker:'Ducati',
            engine:'937cc'
};
bike.wheelType = 'Alloy';
console.log(bike.wheelType);   //Output: Alloy
```

Object methods:

Behavior can be added to the object as well, behaviors are nothing but functions or methods. Methods can be part of object while creation or can be added later like properties as shown below

```
let bike = {
  name: 'SuperSport',
  maker:'Ducati',
  start: function() {
     console.log('Starting the engine...');
  }
};
bike.start();   //Output: Starting the engine...
/* Adding method stop() later to the object */

bike.stop = function() {
   console.log('Applying Brake...');
```

```
}

bike.stop();    //Output: Applying Brake...
```

Create JavaScript Object with Constructor

Constructor is nothing but a function and with help of new keyword, constructor function allows to create multiple objects of same flavor as shown below

```
function Vehicle(name, maker) {
    this.name = name;
    this.maker = maker;
}
let car1 = new Vehicle("Fiesta");
let car2 = new Vehicle("Santa Fe");
console.log(car1.name);    //Output: Fiesta
console.log(car2.name);    //Output: Santa Fe
```

this and new keyword :

a) Every function, while executing has a reference to its current execution context called this (keyword).

b) The new keyword in front of any function turns the function call into constructor call

Create JavaScript Object with create method

Object.create() allowed to create object with more atrribute options like configurable, enumerable, writable and value as shown below

```
let car = Object.create(Object.prototype,
 {
  name:{
   value: 'Fiesta',
   configurable: true,
   writable: true,
   enumerable: false
  },
  maker:{
   value: 'Ford',
   configurable: true,
   writable: true,
   enumerable: true
  }
 });
console.log(car.name)   //Output: Fiesta
```


prototype:

a) Every single object is built by constructor function.

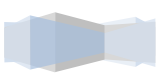
b) A constructor function makes an object linked to its own prototype.

c) Prototype is an arbitrary linkage between the constructor function and object.

Create JavaScript Object using ES6 classes

ECMAScript 6 (newer version of javascript) supports class concept like any other Statically typed or object oriented language. So, object can be created out of a class in javascript as well as shown below

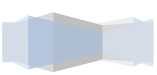
```
class Vehicle {

 constructor(name, maker, engine) {

            this.name = name;
            this.maker =  maker;
            this.engine = engine;
   }

}
            let bike1 = new Vehicle('Hayabusa', 'Suzuki', '1340cc');
            let bike2 = new Vehicle('Ninja', 'Kawasaki', '998cc');
            console.log(bike1.name);    //Output: Hayabusa
            console.log(bike2.maker);   //Output: Kawasaki
```

Methods to the JavaScript Class

Methods can be part of class while declaration or can be added later to the created object as shown below

```
class Vehicle {

 constructor(name, maker, engine) {

            this.name = name;
            this.maker =  maker;
            this.engine = engine;
 }
 start() {
  console.log("Starting...");
 }
}
 let bike = new Vehicle('Hayabusa', 'Suzuki', '1340cc');
 bike.start();    //Output: Starting...
/* Adding method brake() later to the created object */

bike.brake = function() {
            console.log("Applying Brake...");
```

```
}

bike.brake();   //Output: Applying Brake...
```

topic 3 Map

JavaScript has always had a very spartan standard library. Sorely missing was a data structure for mapping values to values. The best you can get in ECMAScript 5 is a map from strings to arbitrary values, by abusing objects. Even then there are several pitfalls that can trip you up

The Map data structure in ECMAScript 6 lets you use arbitrary values as keys and is highly welcome.

Basic operations

```
let map = new Map();


map.set('foo', 123);


console.log( map.get('foo'));
//123


map.has('foo')
console.log(map.has('foo'));
//true
map.delete('foo')
//true
```

```
map.has('foo')
//false
```

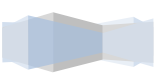
Exmpleles:

```
let map = new Map();
map.set('foo', true);
map.set('bar', false);

map.size
//2
map.clear();
map.size
//0
```

Setting up a map

You can set up a map via an iterable over key-value “pairs” (arrays with 2 elements). One possibility is to use an array (which is iterable):

```
let map = new Map([
    [ 1, 'one' ],
    [ 2, 'two' ],
    [ 3, 'three' ], // trailing comma is ignored
]);
```

1) Alternatively, the set method is chainable:

```
let map = new Map()
.set(1, 'one')
.set(2, 'two')
.set(3, 'three');
```

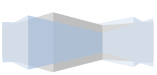
Keys:

What keys are considered equal?

Most map operations need to check whether a value is equal to one of the keys. They do so via the internal operation SameValueZero, which works like === [1], but considers NaN to be equal to itself.

```
NaN === NaN
//false
```

```
let map = new Map();

map.set(NaN, 123);
map.get(NaN)
//123
```

```
map.set(-0, 123);

console.log(map.get(+0))

//123
```

Different objects are always considered different. That is something that can't be configured (yet), as explained later

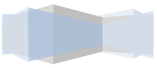
```
new Map().set({}, 1).set({}, 2).size

//2
```

Getting an unknown key produces undefined:

```
new Map().get('asfddfsasadf')

//undefined
```

Iterating:

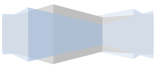
```
let map = new Map([
   [false, 'no'],
   [true,  'yes'],
]);

for (let key of map.keys()) {
            console.log(key);
}
```

```
let map = new Map([
   [false, 'no'],
   [true,  'yes'],
]);

 for (let key of map.keys()) {
              console.log(key);
}
for (let value of map.values()) {
              console.log(value);
}
for (let entry of map.entries()) {
   console.log(entry[0], entry[1]);
}

// Output:
// false no
// true yes
```

Destructuring enables you to access the keys and values directly:

```
for (let [key, value] of map.entries()) {
   console.log(key, value);
```

```
}
```

false "no"

true "yes"

The default way of iterating over a map is entries():

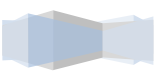
```
for (let [key, value] of map) {
   console.log(key, value);
}
```

false "no"

true "yes"

Spreading iterables

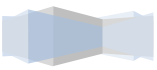
The spread operator (...) turns an iterable into the arguments of a function or parameter call. For example, Math.max() accepts a variable amount of parameters. With the spread operator, you can apply that method to iterables.



```
let arr = [2, 11, -1];

Math.max(...arr)

//11


let map = new Map([
   [1, 'one'],
   [2, 'two'],
   [3, 'three'],
]);
let arr = [...map.keys()]; // [1, 2, 3]
```

Looping over entries:

```
let map = new Map([
   [false, 'no'],
   [true,  'yes'],
]);
map.forEach((value, key) => {
   console.log(key, value);
});
// Output:
// false no
// true yes
```

Basic operations

```
let set = new Set();
 set.add('red')

 set.has('red')
//true
 set.delete('red')
//true
 set.has('red')
false
```

Determining the size of a set and clearing it:

```
 let set = new Set();
set.add('red')
set.add('green')

 set.size
//2
 set.clear();
 set.size
```

```
//0
```