Intro To JavaScript – Day 2

Objects

The only primitive values in JavaScript are: `string`, `number`, `boolean` (`true`, `false`), `null`, and `undefined`. Everything else is an object. But, strings, numbers, and booleans behave like immutable objects.

An object is an unordered collection of properties, with each property having a name and a value. Each property name is a string.

At one level, you can think of an object as a simple associative array (or hash table) of name:value pair mappings. However, an object is more than just that.

Objects are dynamic, properties can be added and deleted at runtime.

- Object Properties
  - Name: Can be any valid string, including having spaces
  - Three property attributes associated with every property
    - writable : Can the value be set.
    - enumerable : Specifies if the property is returned by a for/in loop.
    - configurable : Specifies if the property can be deleted and its attributes altered.
  - Three object attributes
    - prototype : reference to another object, from which properties are inherited.
    - class : a string that categorizes the type of the object.
    - extensible : whether new properties can be added to the object.
- “own” property used to describe a property defined directly on that object (not from its prototype).
- “inherited” property used to describe a property defined by an object's prototype.

Object Literals

```
let myObject = {};
let point2d = {
    x: 0,
    y: 1
};
let point3d = {
    x: 0,
    y: 1,
    z: 2
};
console.log(point2d.x + ', ' + point2d.y);
console.log(point3d.x + ', ' + point3d.y + ', ' + point3d.z);
```

Compound Objects

```
var person = {
    name: {
        first: 'John',
        last: 'Doe'
    },
    address: {
        line1: '1000 West 1000 North',
        city: 'Logan',
        state: 'UT',
        'zip-4': '84321-1024'
    },
    age: 26
};
console.log(person.name.first);
console.log(person.address.city);
console.log(person.address['zip-4']);
```

Property Getters & Setters

If you don't do anything special, a property is both read and write. If this is what you want, don't need to do anything special.

On the other hand, if you want to control access to the values (read or write) to a property, you can use the `get` and `set` accessor properties to do something different. For example...

```
var circle = {
    radius: 4,
    get diameter() { return this.radius * 2; },
    set diameter(value) { this.radius = value / 2; }
};
console.log(circle.radius);
console.log(circle.diameter);
circle.diameter = 4;
console.log(circle.radius);
console.log(circle.diameter);
```

When we enumerate the properties both `radius` and `diameter` show up...

```
for (var property in circle) {
    console.log(property);
}
```

Object.defineProperty

How about another approach to defining a property, we want to control it in more detail...

```
let circle = {
    radius: 4
};
Object.defineProperty(circle, 'diameter', {
    value: circle.radius * 2,
    writable: false,
    enumerable: true,
    configurable: true
});
console.log(circle.radius);
console.log(circle.diameter);
circle.diameter = 4;
console.log(circle.radius);
console.log(circle.diameter);
```

If we enumerate this, once again, both `radius` and `diameter` show up. But if we change the `enumerable` setting to `false`, it doesn't show up (demonstrate this).

Can use `get` and `set` functions instead of the `value` property…

```
Object.defineProperty(circle, 'diameter', {
    get() { return circle.radius * 2; },
    set(diameter) { circle.radius = diameter / 2; },
    writable: false,
    enumerable: true,
    configurable: true
});
```

Deleting Object Properties

Using the above example, lets take a quick look at the ability to delete properties from an object at runtime.

```
console.log('--- before ---');
for (let property in circle) {
    console.log(property);
}

delete circle.diameter;

console.log('--- after ---');
for (let property in circle) {
    console.log(property);
}
```

Show how changing the `configurable` property to `false` prevents the property from being deleted.

Object Prototypes

JavaScript doesn't have classes, and therefore, no class-based inheritance. But yet, we say that JavaScript is object oriented, including inheritance, how does that work? Prototypes!

(nearly) Every JavaScript object has a second object associated with it, a prototype. In a sense, the first object inherits all properties of the prototype; it is better to say that it is linked to the prototype rather than inheriting from it.

All objects created through literals have the same prototype, referred to as `Object.prototype`. `Object.prototype` itself does not have a prototype, one of the few objects without a prototype.

This is all better understood with an example. Let's define a 2D Point and then use it as the basis for creating a 3D point. Here it is...

```
var point2d = {
    x: 0,
    y: 1
};
var point3d = Object.create(point2d);
point3d.z = 2;
console.log('--- point2d ---');
for (var property in point2d) {
    console.log(property);
}
console.log('--- point3d ---');
for (var property in point3d) {
    console.log(property);
}
console.log('--- point3d own properties ---');
for (var property in point3d) {
    if (point3d.hasOwnProperty(property)) {
        console.log(property);
    }
}
```

`Object.create` is used to create a new object using the specified object as the prototype. *Draw a diagram that illustrates this.*

At this point, do some playing around with the properties of `point2d` *and* `point3d` *to show how the prototype object and properties work. Here are some possibilities to try...*

```
point2d.name = 'Point2d';
console.log('--- point2d ---');
for (var property in point2d) {
    console.log(property);
}
console.log('--- point3d ---');
for (var property in point3d) {
    console.log(property);
}
point3d.name = 'Point3d';
console.log(point2d.name);
console.log(point3d.name);
console.log('--- point3d own properties ---');
for (var property in point3d) {
    if (point3d.hasOwnProperty(property)) {
        console.log(property);
        }
    }
```

Here is something to build towards:

- Add the `report` function
- Add each of these following one at a time to show how the prototype inheritance builds up over time:
    - `point2d.x = 10;`
    - `point3d.x = 20;`
    - `point3d.y = 30;`

```
function report(point) {
    for (let property in point) {
        let message = property + ' : ' + point[property];
        if (point.hasOwnProperty(property)) {
            message = '(own) ' + message;
        }
        console.log(message);
    }
}

let point2d = {
    x: 0,
    y: 1
};

let point3d = Object.create(point2d);
point3d.z = 2;

point2d.x = 10;
point3d.x = 20;
point3d.y = 30;

point2d.name = 'Point2d';
point3d.name = 'Point3d';

console.log('--- Point2d ---');
report(point2d);

console.log('--- Point3d ---');
report(point3d);
```

Augmenting Prototypes

It is possible to augment existing types with new functions. This is done by changing the prototype of the type you wish to augment. Take a look at this...

```
let message = 'This is a message';
let message2 = 'This is a different message';

String.prototype.toL337 = function() {
    return this.replace(/e/g, '3').replace(/i/g, '1').replace(/T/g, '7');
};

console.log(message.toL337());
console.log(message2.toL337());
```

We can see that `String` has been augmented with a new `toL337()` function and can be called from `String` object!