Interactive Web Development

Objects

Dr Russ Ross

Dixie State University—Computer and Information Technologies

Spring 2016
Objects

**Reading:** *JavaScript: The Good Parts*, Chapter 3
Objects

The simple types of JavaScript are:

- numbers
- strings
- booleans (true and false)
- null
- undefined

The three types marked in bold are object-like; they have methods, but they are immutable. These types are all true objects:

- arrays
- functions
- regular expressions
- objects
Objects

An object is a container of properties:

- a property has a name and a value
- a property name can be any string, including the empty string
- a property value can be any JavaScript value except undefined
- there is no constraint on the names of new properties
- there is no constraint on the values of properties
- objects can contain other objects, making it easy to represent tree and graph structures
- objects are useful for collecting and organizing data

Objects are class-free. They have a prototype linkage feature that allows one object to inherit the properties of another. When done right, this makes object initialization quick and reduces memory consumption.
Object literals are a convenient notation for creating new object values:

```javascript
var empty_object = {};

var stooge = {
    "first-name": "Jerome",
    "last-name": "Howard"
};
```

An object literal is a pair of curly braces surrounding zero or more name/value pairs. It can appear anywhere an expression can appear.
Object literals

Commas separate the pairs. A property’s value can come from any expression, including another object literal. Objects can nest:

```javascript
var flight = {
    airline: "Oceanic",
    number: 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
    arrival: {
        IATA: "LAX",
        time: "2004-09-23 10:42",
        city: "Los Angeles"
    }
};
```

Quotes are optional for property names that are legal JavaScript names and not reserved words, e.g., `first_name` vs. "first-name".
Values can be retrieved from an object by wrapping a string expressing in a [ ] suffix:

```
stooge["first-name"]  // "Joe"
flight.departure.IATA  // "SYD"
```

If the string expression is a constant, and it is a legal JavaScript name and is not a reserved word, then you can use the . notation instead. This is more compact and easier to read.

The `undefined` value is produced when you attempt to retrieve a nonexistent member:

```
stooge["middle-name"]  // undefined
flight.status           // undefined
stooge["FIRST-NAME"]   // undefined
```
The `||` operator can be used to fill in default values:

```javascript
var middle = stooge["middle-name"] || "(none)";
var status = flight.status || "unknown";
```

Attempting to retrieve values from `undefined` will throw a `TypeError` exception. The `&&` operator can guard against this:

```javascript
flight.equipment // undefined
flight.equipment.model // throw "TypeError"
flight.equipment && flight.equipment.model // undefined
```
Update

A value in an object can be updated by assignment. If the property name already exists in the object, the property value is replaced:

```javascript
stooge['first-name'] = 'Jerome';
```

If the object does not already have that property name, the object is augmented:

```javascript
stooge['middle-name'] = 'Lester';
stooge.nickname = 'Curly';
flight.equipment = {
    model: 'Boeing 777'
};
flight.status = 'overdue';
```
Objects are passed by reference. They are never copied:

```javascript
var x = stooge;
x.nickname = 'Curly';
var nick = stooge.nickname;
    // nick is 'Curly' because x and stooge
    // are references to the same object

var a = {}, b = {}, c = {};
    // a, b, and c each refer to a
    // difference empty object

a = b = c = {};
    // a, b, and c all refer to
    // the same empty object
```
Prototype

Every object is linked to a prototype object from which it can inherit properties. All objects created from object literals are linked to Object.prototype, an object that comes standard with JavaScript.

The mechanism to link a new object to a specific prototype is messy. The following code simplifies it:

```javascript
if (typeof Object.beget !== 'function') {
    Object.beget = function (o) {
        var F = function () {};
        F.prototype = o;
        return new F();
    };
}
```

With that, you can use:

```javascript
var another_stooge = Object.beget(stooge);
```
Prototype

The prototype link has no effect on updating:

```
another_stooge['first-name'] = 'Harry';
another_stooge['middle-name'] = 'Moses';
another_stooge.nickname = 'Moe';
```

Changes to an object do not change the prototype object. Updates to the prototype object are immediately reflected in the objects linked to that prototype:

```
stooge.profession = 'actor';
another_stooge.profession // 'actor'
```

When you try to retrieve a property, the object is first searched. If the property is missing, the prototype object is searched, and so on, eventually ending with `Object.prototype`. This process is called **delegation**. If the property is not found, the result is `undefined`. 
It is easy to inspect an object to determine what properties it has by attempting to retrieve the properties and examining the values obtained:

```javascript
typeof flight.number // 'number'
typeof flight.status // 'string'
typeof flight.arrival // 'object'
typeof flight.manifest // 'undefined'
```

The `typeof` operator is very useful for this task.
Reflection

Any property on the prototype chain can produce a value:

```javascript
typeof flight.toString // 'function'
typeof flight.constructor // 'function'
```

One solution is to ignore all functions. When reflecting, you are usually looking for data.

You can also use the `hasOwnProperty` method:

```javascript
flight.hasOwnProperty('number') // true
flight.hasOwnProperty('constructor') // false
```

It returns `true` if the object has a particular property. `hasOwnProperty` does not look at the prototype chain.
The `for in` statement can loop over all of the property names in an object. All properties will be included; functions and prototype properties that you may not be interested in will be part of the enumeration. Normally, you should filter these out:

```javascript
var name;
for (name in another_stooge) {
    if (typeof another_stooge[name] !== 'function') {
        document.writeln(name + ': ' + another_stooge[name]);
    }
}
```

Using `hasOwnProperty` is also commonly used to filter out undesirable properties.
The order in which properties are enumerated with `for in` is not guaranteed. In particular, it is unlikely to match the order in which the properties were created.

If you need properties to appear in a certain order, `for in` may not be the right tool:

```javascript
var i;
var properties = ['first-name', 'middle-name', 'last-name', 'profession'];
for (i = 0; i < properties.length; i += 1) {
    document.writeln(properties[i] + ': ' + another_stooge[properties[i]]);
}
```
The `delete` operator can be used to remove a property from an object:

- it will remove a property from the object if it has one
- it will not touch the prototype chain
- removing a property may allow a property from the prototype linkage to show through

```javascript
another_stooge.nickname // 'Moe'

// Remove nickname from another_stooge, revealing
// the nickname of the prototype
delete another_stooge.nickname;

another_stooge.nickname // 'Curly'
```
Global abatement

It is easy to create and use global variables in JavaScript. All global variables are actually properties of the `window` object, and overusing this can make your programs fragile.

One way to minimize the danger is to create a single global variable for your application:

```javascript
var MYAPP = {};
```

Then you can use that variable as a container for your application:

```javascript
MYAPP.stooge = {
  "first-name": "Joe",
  "last-name": "Howard"
};

MYAPP.flight = {
  airline: "Oceanic",
  // ...
};
```