Topic 4 - JavaScript

Introduction to JavaScript

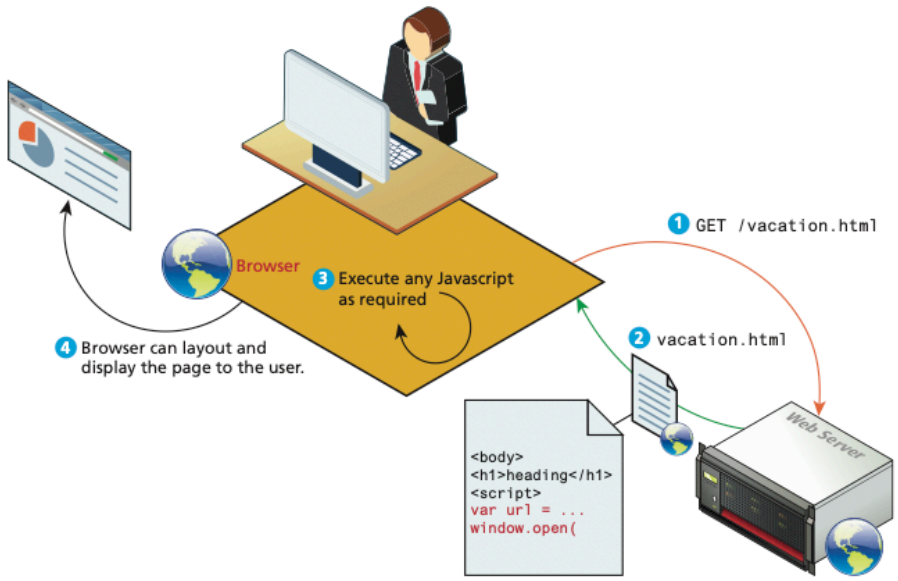
- JS Syntax, datatypes
- Inputs/Outputs
- Basic Control Structures, Arrays
- Objects and Functions
- The DOM
- Events
- JS libraries and Ajax

Language Fundamentals

What is JS?

What is JavaScript & What Can It Do?

Client-Side Scripting



What is JavaScript & What Can It Do?

JavaScript's History

- JavaScript was introduced by Netscape in their Navigator browser back in 1996
- JavaScript that is supported by your browser contains language features
  - not included in the current ECMAScript specification and
  - missing certain language features from that specification

The latest version of ECMAScript is the Ninth Edition (generally referred to as ES9 or ES2018).

What is JavaScript & What Can It Do?

JavaScript and Web 2.0

- Early JavaScript had only a few common uses:
- 2000s onward saw more sophisticated uses for JavaScript
- **AJAX** as both an acronym and a general term

Where Does JavaScript Go?

Inline JavaScript

Inline JavaScript refers to the practice of including JavaScript code directly within certain HTML attributes

```
<a href="JavaScript:OpenWindow();">more info</a>

<input type="button" onClick="alert('Are you sure?');" />
```

Where Does JavaScript Go?

Embedded JavaScript

Embedded JavaScript refers to the practice of placing JavaScript code within a <script> element

```
<script type="text/javascript">
        /* A JavaScript Comment */
        alert("Hello World!");
</script>
```


Where Does JavaScript Go?

External JavaScript

external JavaScript files typically contain function definitions, data definitions, and entire frameworks.

```
<head>
    <script type="text/javascript" src="greeting.js"></script>
</head>
```

Where Does JavaScript Go?

Users without JavaScript

- Web crawler
- Browser plug-in.
- Text-based client.
- Visually disabled client.
- The <NoScript> Tag

Variables, Data Types, Outputs

Variables and Data Types

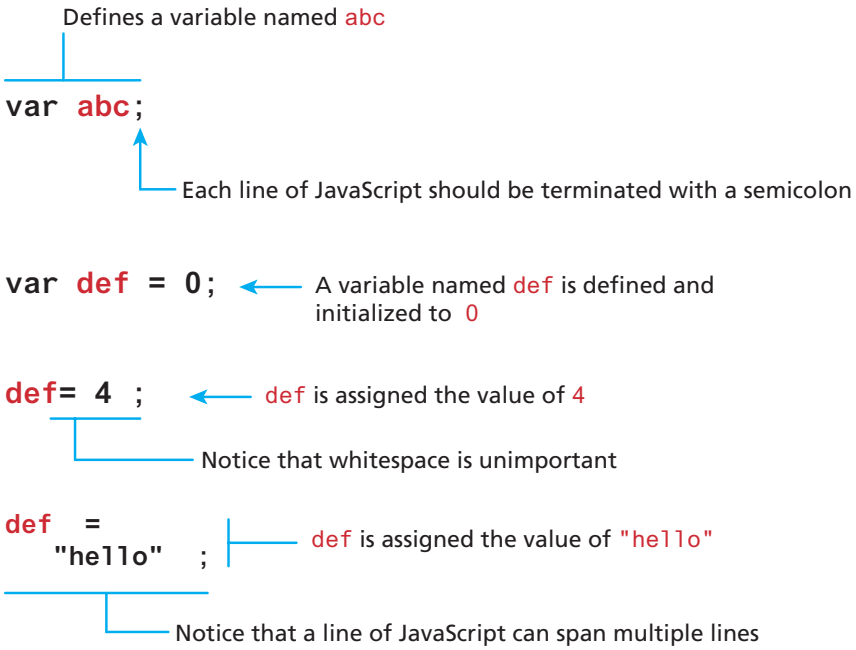
Variables in JavaScript are **dynamically typed**

This simplifies variable declarations, since we do not require the familiar data-type identifiers

Instead, we simply use the **var** keyword

Variables and Data Types

Example variable declarations and Assignments



Variables and Data Types

Data Types

two basic data types:

- reference types (usually referred to as objects) and
- primitive types

Primitive types represent simple forms of data.

- **Boolean, Number, String, …**

Variables and Data Types

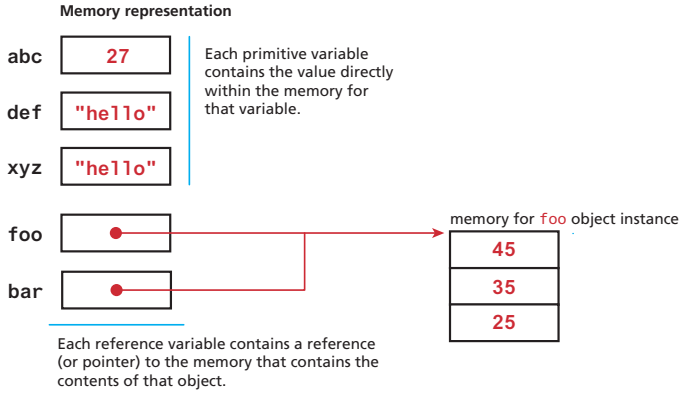
Reference Types

```
var abc = 27;
var def = "hello";
```
variables with primitive types

```
var foo = [45, 35, 25];
```
variable with reference type (i.e., array object)

```
var xyz = def;
var bar = foo;
```
these new variables differ in important ways (see below)

```
bar[0] = 200;
```
changes value of the first element of array



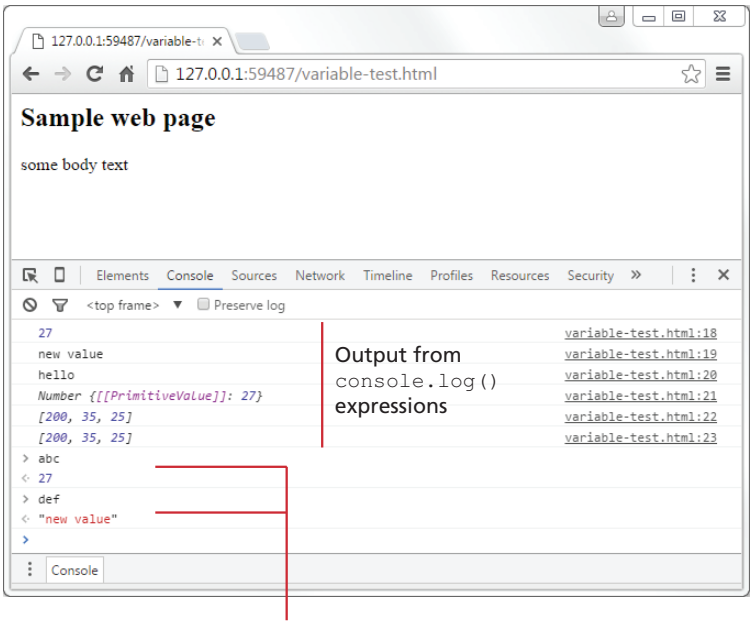
Inputs/Outputs

JavaScript Output

- alert() Displays content within a pop-up box.
- console.log() Displays content in the Browser's JavaScript console.
- document.write() Outputs the content (as markup) directly to the HTML document.

JavaScript Output

Chrome JavaScript Console



Language Constructs

Conditionals, Loops, Arrays

Conditionals

If, else if, else

```
if (hourOfDay > 4 && hourOfDay < 12) {
        greeting = "Good Morning";
}
else if (hourOfDay >= 12 && hourOfDay < 18) {
        greeting = "Good Afternoon";
}
else {
        greeting = "Good Evening";
}
```


Conditionals

Conditional Assignment

```
/* x conditional assignment */
x = (y==4) ? "y is 4" : "y is not 4";
```

Condition — Value if true — Value if false

```
/* equivalent to */
if (y==4) {
   x = "y is 4";
}
else {
   x = "y is not 4";
}
```

Conditionals

Truthy and Falsy

In JavaScript, a value is said to be **truthy** if it translates to true, while a value is said to be **falsy** if it translates to false.

- Almost all values in JavaScript are truthy
- false, null, "", '', 0, NaN, and undefined are falsy

Loops

While and do . . . while Loops

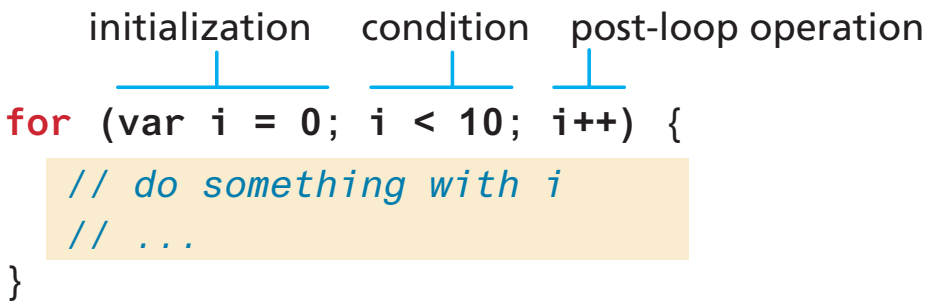
```
var count = 0;

while (count < 10) {
        // do something
        // ...
        count++;
}

count = 0;
do {
        // do something
        // ...
        count++;
} while (count < 10);
```

Loops

For Loops



Arrays

Arrays are one of the most commonly used data structures in programming.

JavaScript provides two main ways to define an array.

- object literal notation
- use the Array() constructor

Arrays

object literal notation

The literal notation approach is generally preferred since it involves less typing, is more readable, and executes a little bit quicker

```
var years = [1855, 1648, 1420];

var countries = ["Canada", "France",
                 "Germany", "Nigeria",
                 "Thailand", "United States"];

var mess = [53, "Canada", true, 1420];
```

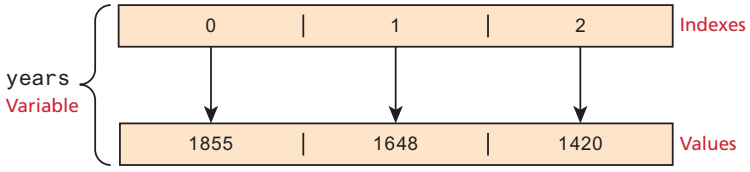

Arrays

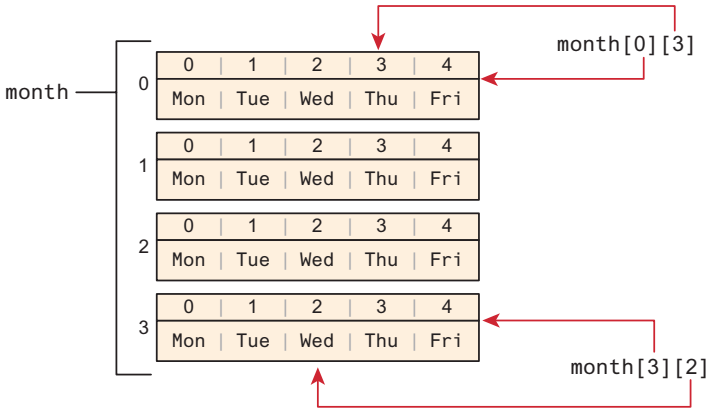
Some common features

- arrays in JavaScript are zero indexed
- [] notation for access
- .length gives the length of the array
- .push()
- .pop()
- concat(), slice(), join(), reverse(), shift(), and sort()

Arrays

Arrays Illustrated





Objects and Functions

Objects

Object Creation and Access—Object Literal Notation

```
var objName = {
        name1: value1,
        name2: value2,
        // ...
        nameN: valueN
};
```

Access using either of:

- objName.name1
- objName["name1"]

Objects

Object Creation—Constructed Form

```
// first create an empty object
var objName = new Object();
// then define properties for this object
objName.name1 = value1;
objName.name2 = value2;
```

Functions

Function Declarations vs. Function Expressions

Functions are the building block for modular code in JavaScript.

```
function subtotal(price,quantity) {

        return price * quantity;

}
```

The above is formally called a **function declaration**, called or invoked by using the () operator

```
var result = subtotal(10,2);
```


Functions

Function Declarations vs. Function Expressions

```
// defines a function using a function expression

var sub = function subtotal(price,quantity) {

        return price * quantity;

};

// invokes the function

var result = sub(10,2);
```

It is conventional to leave out the function name in function expressions

Functions

Anonymous Function Expressions

```
// defines a function using an anonymous function
expression

var calculateSubtotal = function (price,quantity) {

        return price * quantity;

};

// invokes the function

var result = calculateSubtotal(10,2);
```


Functions

Nested Functions

```
function calculateTotal(price,quantity) {

        var subtotal = price * quantity;

        return subtotal + calculateTax(subtotal);

        // this function is nested

        function calculateTax(subtotal) {

                var taxRate = 0.05;

                var tax = subtotal * taxRate;

                return tax;

        }

}
```

Functions

Hoisting in JavaScript

Function declaration is **hoisted** to the beginning of its scope

```
function calculateTotal(price,quantity) {
    var subtotal = price * quantity;
    return subtotal + calculateTax(subtotal);

    function calculateTax(subtotal) {
        var taxRate = 0.05;
        var tax = subtotal * taxRate;
        return tax;
    }
}
```

Variable declaration is hoisted to the beginning of its scope

BUT

Variable assignment is **not** hoisted

```
function calculateTotal(price,quantity) {
    var subtotal = price * quantity;
    return subtotal + calculateTax(subtotal);

    var calculateTax = function (subtotal) {
        var taxRate = 0.05;
        var tax = subtotal * taxRate;
        return tax;
    };
}
```

THUS

The value of the `calculateTax` variable here is undefined

Functions

Callback Functions

```
var calculateTotal = function (price, quantity, tax) {
    var subtotal = price * quantity;
    return subtotal + tax(subtotal);
};

var calcTax = function (subtotal) {
    var taxRate = 0.05;
    var tax = subtotal * taxRate;
    return tax;
};

var temp = calculateTotal(50,2,calcTax);
```

2 The local parameter variable `tax` is a reference to the `calcTax()` function

1 Passing the `calcTax()` function object as a parameter

We can say that `calcTax` variable here is a callback function

Functions

Callback Functions

Passing an anonymous function definition as a callback function parameter

```
var temp = calculateTotal( 50, 2,
                           function (subtotal) {
                              var taxRate = 0.05;
                              var tax = subtotal * taxRate;
                              return tax;
                           }
                         );
```


Functions

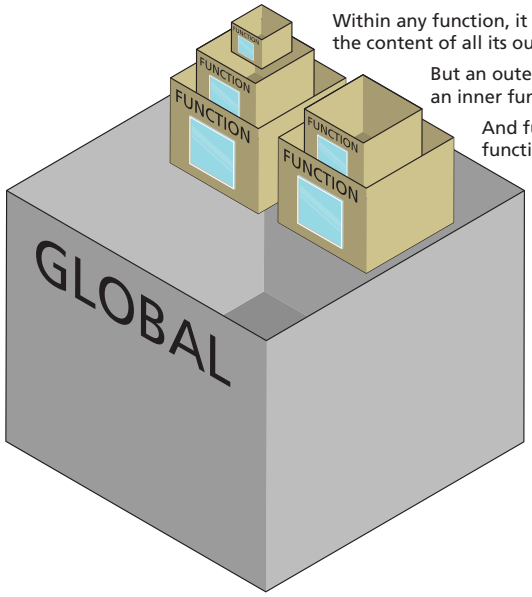
Objects and Functions Together

```
var order = {
    salesDate : "May 5, 2017",
    product : {
        type: "laptop",
        price: 500.00,
        output: function () {
            return this.type + ' $' + this.price;
        }
    },
    customer : {
        name: "Sue Smith",
        address: "123 Somewhere St",
        output: function () {
            return this.name + ', ' + this.address;
        }
    },
    output: function () {
            return 'Date' + this.salesDate;
    }
};
```

Functions

Scope in JavaScript

Each function is like a box
with a one-way window



Within any function, it can see out at the content of all its outer boxes

But an outer function can't look into an inner function

And functions can't see into other functions at the same level

All functions can see anything within global scope

Scope ends at global ... functions can't see outside of the global box

Functions

Scope in JavaScript

Anything declared inside this block is global and accessible everywhere in this block

global variable c is defined
global function `outer()` is called

```
1  var c = 0;
2  outer();
```

Anything declared inside this block is accessible everywhere within this block

```
function outer() {
```

Anything declared inside this block is accessible only in this block

```
  function inner() {
```

local (outer) variable a is accessed

```
5     console.log(a);
```

✓ allowed — outputs 5

local (inner) variable b is defined

```
6     var b = 23;
```

global variable c is changed

```
7     c = 37;
```

✓ allowed

```
  }
```

local (outer) variable a is defined

```
3   var a = 5;
```

local function `inner()` is called

```
4   inner();
```

global variable c is accessed

```
8   console.log(c);
```

✓ allowed — outputs 37

undefined variable b is accessed

```
9   console.log(b);
```

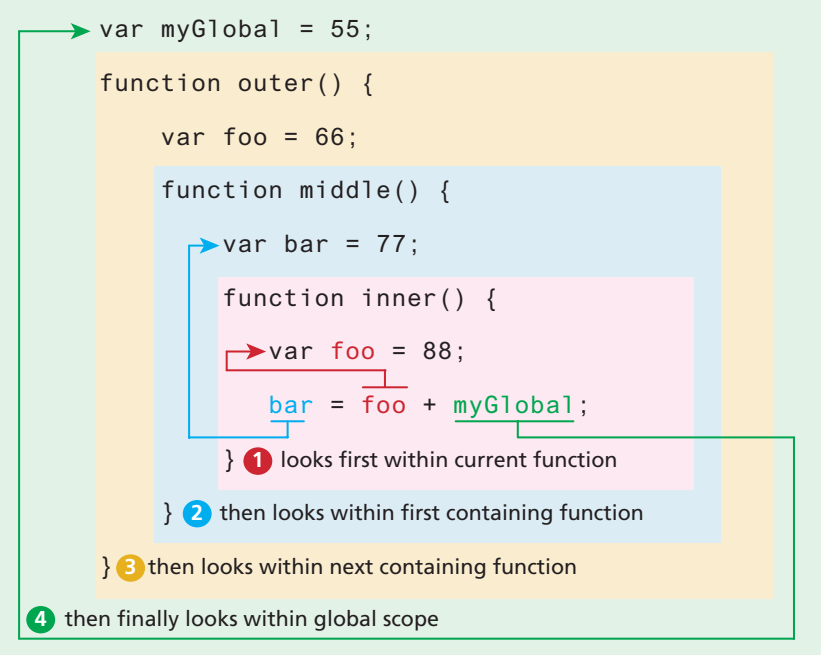
✖ not allowed — generates error or outputs undefined

```
}
```

Functions

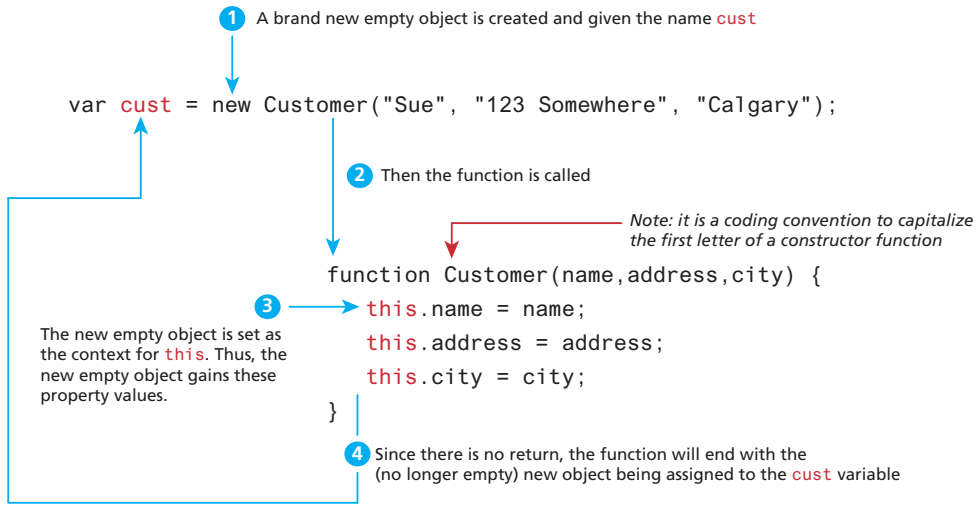
Scope in JavaScript

Remember that scope is determined at design-time



Functions

Function Constructors



Object Prototypes

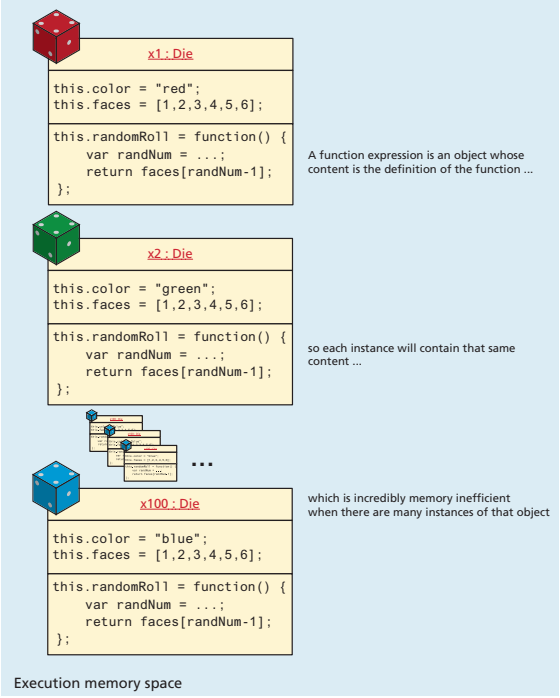
There's a better way

While the constructor function is simple to use, it can be an inefficient approach for objects that contain methods.

Prototypes are an essential syntax mechanism in JavaScript, and are used to make JavaScript behave more like an object-oriented language.

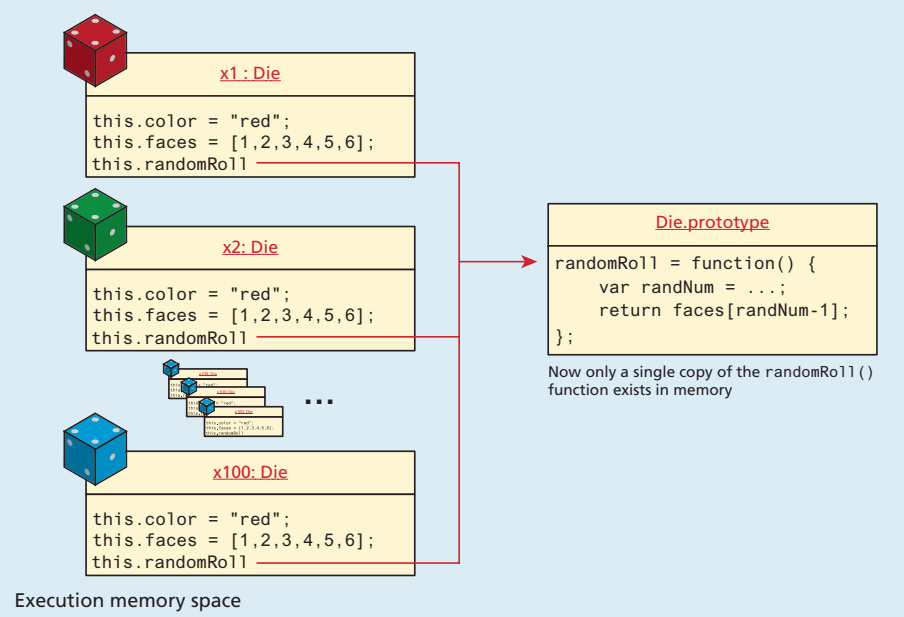
Object Prototypes

Methods get duplicated…



Object Prototypes

Using Prototypes reduces duplication at run time.



Object Prototypes

Using Prototypes to Extend Other Objects

```
String.prototype.countChars = function (c) {
        var count = 0;
        for (var i = 0; i < this.length; i++) {
                if (this.charAt(i) == c)
                        count++;
                }
        return count;
}
var msg = "Hello World";
console.log(msg + "has" + msg.countChars("l") + " letter l's");
```