

Javascript: Intro

CITS3403 Agile Web Development

Unit Coordinator: Tim French

2023 Semester 1

JavaScript

JavaScript is a high-level, dynamic, untyped, and interpreted programming language. It has been standardized in the ECMAScript language specification. Alongside HTML and CSS, it is one of the three essential technologies of World Wide Web content production. JavaScript is prototype-based with first-class functions, making it a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles.

- Language specification: <http://www.ecmascript.org/>
- Tutorial: <http://www.w3schools.com/js/>

Components

- Core
 - The heart of the language
- Client-side
 - Library of objects supporting browser control and user interaction
- Server-side
 - Library of objects that support use in web server

Uses of JavaScript

- Provide alternative to server-side programming
 - Servers are often overloaded
 - Client processing has quicker reaction time
- JavaScript can work with forms
- JavaScript can interact with the internal model of the web page (“Document Object Model” - more on this soon...)
- JavaScript is used to provide more complex user interface than plain forms with HTML/CSS can provide
- JQuery is one of the most popular development libraries.
- Node is a server-side javascript environment
- Linux in javascript??? <http://jslinux.org/>

Event-Driven Computation

- Users actions, such as mouse clicks and key presses, are referred to as *events*
- The main task of many JavaScript programs is to respond to events
- For example, a JavaScript program could validate data in a form before it is submitted to a server
 - *Caution:* It is important that crucial validation be done by the server. It is relatively easy to bypass client-side controls
 - For example, a user might create a copy of a web page but remove all the validation code.

Edit This Code:

See Result »

Result:

```
<!DOCTYPE html>
<html>
<body>

<button onclick="getElementById('demo').innerHTML=Date()">The time is?
</button>

<p id="demo"></p>

</body>
</html>
```

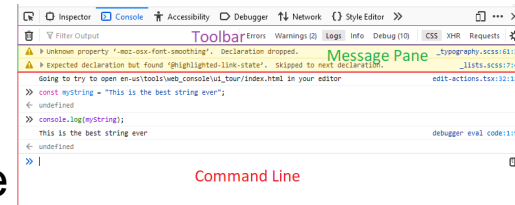
The time is?

Sun Mar 20 2016 13:57:31 GMT+0800 (AWST)

Javascript execution environments

There are two main execution environments for JavaScript:

- The browser: every modern web browser is able to execute javascript, and many javascript functions refer explicitly to an HTML container or window. To test and execute Javascript, you need a html file to call the javascript function, and a browser to open that file.



- NodeJS: Node is a server side javascript environment. This is useful since we can run the same code the client uses on the server. This is more like a tradition console environment you may have seen (eg, python).



- You can install Node on your local machine from <https://nodejs.org/en/>



HTML/JavaScript Documents

There are several ways to include javascript in a web-page:

- Including the code in the head, inside a `script` tag.
- Including the code inside the body, inside a `script` tag.
- Providing a url to an external file containing the code.

```
<!DOCTYPE html>
<html>

<head>
<script>
function myFunction() {
    document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>
</head>
<body>

<h1>A Web Page</h1>
<p id="demo">A Paragraph</p>
<button type="button" onclick="myFunction()">Try it</button>

</body>
</html>
```

JavaScript in Head

Paragraph changed.

Try it

General Syntactic Characteristics

- Identifiers
 - Start with \$, _, letter
 - Continue with \$, _, letter or digit
 - Case sensitive
 - camelCase preferred
- Comments
 - //
 - /* ... */
- Reserved words...

abstract	arguments	boolean	break	byte
case	catch	char	class*	const
continue	debugger	default	delete	do
double	else	enum*	eval	export*
extends*	false	final	finally	float
for	function	goto	if	implements
import*	in	instanceof	int	interface
let	long	native	new	null
package	private	protected	public	return
short	static	super*	switch	synchronized
this	throw	throws	transient	true
try	typeof	var	void	volatile
while	with	yield		

Statements *should* be terminated with a semicolon

The interpreter will insert the semicolon if missing and the statement seems to be complete

Can be a problem:

```
return  
X;
```

Like HTML, the environment will tolerate incorrect code as much as possible.

Data Types

- Javascript has the following data types
 - Numbers
 - Strings
 - Booleans
 - Null
 - Undefined
 - Objects
 - Functions
 - Arrays
 - Date
 - RegExp
 - Math

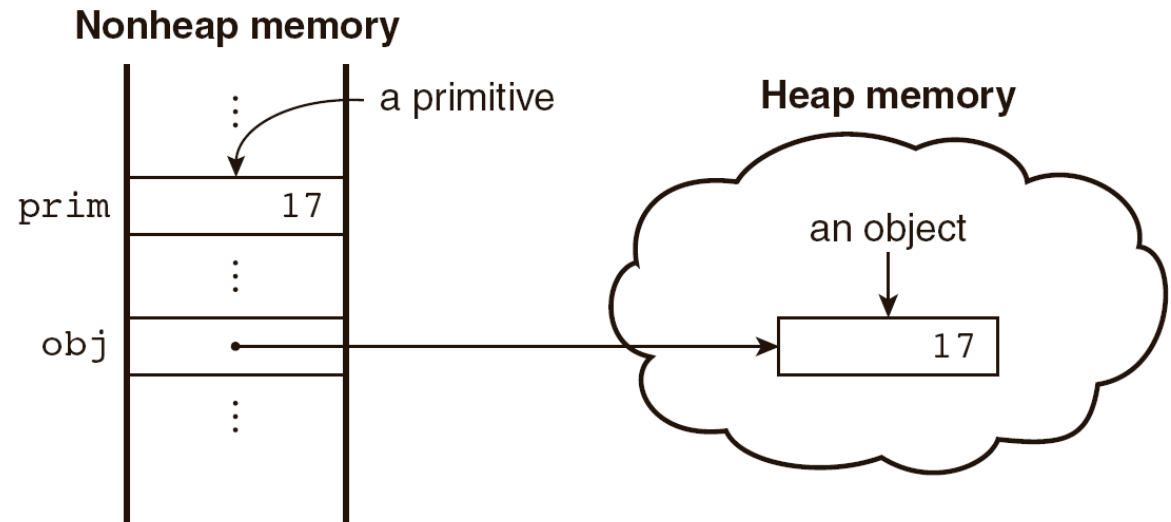


Figure 4.1 Primitives and objects

Numeric Literals

- Number values are represented internally as *double-precision floating-point*
 - There is no integer type in here. According to the spec., they are all *double-precision 64-bit format* IEEE 754 values. So you might have to be a little careful with arithmetic:
 $0.1 + 0.2 = 0.30000000000000004$
 - For advanced mathematical operations you can use the built-in [Math](#) Math object

```
var value = Math.sin(3.5); // gets the sine of 3.5
```
- You can convert a string to an integer by using the [parseInt\(\)](#) function –

```
var i = parseInt("124", 10); // i now contains 124
```
- *NaN (Not a number)* is returned if the argument string to *parseInt()* is non-numeric:

```
var value = parseInt("hello", 10); // value now contains NaN
```
- *NaN* is toxic –as an input to any mathematical operation the result will also be *NaN*

```
var value = NaN + 5; // value is now NaN
```
- You can check for *NaN* by using the built-in [isNaN\(\)](#) function –

```
isNaN(value); // will return true if value is NaN
```
- Javascript also has some special values denoting [Infinity](#) and *–Infinity*

```
var infinity = 1 / 0; // infinity now contains Infinity  
var negativeInfinity = -1 / 0; // as described above
```

The Number Object

- Properties

Property	Meaning
<code>MAX_VALUE</code>	Largest representable number
<code>MIN_VALUE</code>	Smallest representable number
<code>NaN</code>	Not a number
<code>POSITIVE_INFINITY</code>	Special value to represent infinity
<code>NEGATIVE_INFINITY</code>	Special value to represent negative infinity
<code>PI</code>	The value of π

Operations resulting in errors return NaN

- Use `isNaN(a)` to test if `a` is NaN

- `toString` method converts a number to string

Type 2 – Strings

- Strings in Javascript are sequence of Unicode characters, where each character is represented by a 16-bit number. This is a very good news to anyone who has to deal with internationalization.
- A String literal is delimited by either single or double quotes
 - There is no difference between single and double quotes
 - Certain characters may be escaped in strings
 - \' or \" to use a quote in a string delimited by the same quotes
 - \\ to use a literal backslash
 - \n new line
 - \t tab
 - etc
 - The empty string "" or "" has no characters
- They have some useful properties and methods for manipulation like [length](#), [charAt\(\)](#), [replace\(\)](#), [toUpperCase\(\)](#), [toLowerCase\(\)](#) etc.
- Javascript doesn't have any Character data-type. So if you want to represent a single character, you need to use a string of length 1.

String Properties and Methods

- One property: length
 - Note to Java programmers, this is not a method!
- Character positions in strings begin at index 0

Method	Parameters	Result
<code>charAt</code>	A number	Returns the character in the <code>String</code> object that is at the specified position
<code>indexOf</code>	One-character string	Returns the position in the <code>String</code> object of the parameter
<code>substring</code>	Two numbers	Returns the substring of the <code>String</code> object from the first parameter position to the second
<code>toLowerCase</code>	None	Converts any uppercase letters in the string to lowercase
<code>toUpperCase</code>	None	Converts any lowercase letters in the string to uppercase

Other Primitive Types

- Null
 - `null` is a reserved word
 - A variable that is intentionally not assigned a value has a null value
 - Using a null value usually causes an error
- Undefined
 - The value of a variable that is not declared or not assigned a value
- Javascript distinguishes between null, which is a special type of object that indicates a deliberate non-value, and undefined, which is an object of type undefined that indicates an uninitialized value.
- Boolean
 - Two values: `true` and `false`
- Javascript has a boolean type, with possible values of *true* and *false*. Any value can be converted to a boolean according to the following rules –
 - `false`, 0, the empty string, NaN, null, and undefined all become false
 - all other values become true.

Javascript will apply these rules whenever it expects a boolean, but you can coerce these type conversion by using the *Boolean()* function.

Declaring Variables

- JavaScript is *dynamically typed*, that is, variables do not have declared types
 - A variable can hold different types of values at different times during program execution
- A variable is declared using the keywords `var`, `let`, `const` (or nothing)

```
var counter, index, pi = 3.14159265, rover = "Palmer",
stop_flag = true;
const x = 6, y = 7;
let z = x+y;
zz = z;
```
- If a variable remains uninitialized, then its type is undefined.
- An important difference from other languages like Java is that in Javascript, you don't get *block-level* scope, only functions have scope. So if a variable is defined using *var* inside an *if* or *for* block, it will be visible to the entire function.
- *let* and *const* do have block level scope.
- In Javascript, there is no strong type-checking like Java. You can declare a variable to hold an integer and then you can assign a string to that same variable

```
var value = 5;    value = "Hello"; // No error
```

Assignments and Operators


- Plain assignment indicated by =
- Compound assignment with: += -= /= *= %= ...
- `a += 7` means the same as `a = a + 7`
- Like Java, you can use + to concatenate two different strings. You can also use it to convert a string to a number

```
let value = + "123";
```

- Numeric Operators
 - Standard arithmetic
+ * - / %
 - Increment and decrement
-- ++
- String Operators
 - Concatenation
+
- Boolean Operators
 - !, &&, ||

Operators	Associativity
++, --, unary -	Right
*, /, %	Left
+, -	Left
>, <, >=, <=	Left
==, !=	Left
===, !==	Left
&&	Left
	Left
=, +=, -=, *=, /=, &&=, =, %=	Right

Highest-precedence operators are listed first.



Implicit Type Conversion

- JavaScript attempts to convert values in order to be able to perform operations
- Numeric Context
 - `7 * "3"`
 - null is converted to 0 in a numeric context, undefined to NaN
- Logical/Boolean Context
 - 0 is interpreted as a Boolean false, all other numbers are interpreted as true
 - The empty string is interpreted as a Boolean false, all other strings (including "0"!) as Boolean true
 - undefined, NaN and null are all interpreted as Boolean false
- `typeof (x)` returns "number" or "string" or "boolean" for primitive types
- `typeof (x)` returns "object" for an object or null
- Two syntactic forms
 - `typeof x`
 - `typeof (x)`

Comparisons

- Comparisons in Javascript can be made using `>`, `<`, `>=`, `<=`, `==`, `===`, `!=` and `!==` operators. These works for both strings and numbers.
- The `==` operator performs type coercion if you give it two different types

```
"dog" == "dog"      // true
1 == true           // true!
'abc' == ['abc']    // true!!
```

- The `===` operator performs returns true only if both operands are equal, and of the same type.

```
"dog" === "dog"     // true
1 === true          // false
'abc' === ['abc']   // false
```

Control Statements

- A *compound statement* in JavaScript is a sequence of 0 or more statements enclosed in curly braces
- A *control construct* is a control statement including the statements or compound statements that it contains

```
var cars = ["BMW", "Volvo", "Saab", "Ford"];
var i = 0;
var text = "";

for (;cars[i];) {
    text += cars[i] + "<br>";
    i++;
}
```

```
var person = {fname:"John", lname:"Doe", age:25};

var text = "";
var x;
for (x in person) {
    text += person[x];
}
```

```
do {
    text += "The number is " + i;
    i++;
}
while (i < 10);
```

```
if (time < 10) {
    greeting = "Good morning";
} else if (time < 20) {
    greeting = "Good day";
} else {
    greeting = "Good evening";
}
```

Control Structures

- Javascript has for, while, do-while loops just like Java. It also has if-else, switch statements and ternary operator. Switch statements can compare string values. You can also use an expression in the case statement.
- The && and || operators use short-circuit logic, which means whether they will execute their second operand depends on the first. This is useful for checking for null objects before accessing their attributes –

```
// && will return Object if it's null
var property = Object &&
    Object.getProperty();
```

- Or for setting their default values –

```
var name = otherName || "default";
```

- The if-then and if-then-else are similar to that in other programming languages, especially C/C++/Java

```
switch (expression) {
  case value_1:
    // statement(s)
  case value_2:
    // statement(s)
  ...
  [default:
    // statement(s)]
}
```

Object Orientation and JavaScript



- JavaScript is *object-based*
 - JavaScript defines objects that encapsulate both data and processing
 - However, JavaScript does not have the same inheritance nor subtyping (therefore polymorphism) as normal OOP such as Java or C#.
- JavaScript provides *prototype-based inheritance*
 - See, for example this Wikipedia article for a discussion:
http://en.wikipedia.org/wiki/Prototype-based_languages
- Objects are collections of *properties*
- Properties are either *data properties* or *method properties*
 - Data properties are either *primitive values* or *references to other objects*
 - Primitive values are often implemented directly in hardware
 - Method properties are *functions* (more later)
- The *Object* object is the ancestor of all objects in a JavaScript program
 - Object has no data properties, but several method properties

Arrays

- Arrays are lists of elements indexed by a numerical value
- Array indexes in JavaScript begin at 0
- *Arrays can be modified in size even after they have been created*
- *Eg*

```
var index;  
var fruits = ["Banana", "Orange", "Apple", "Mango"];  
for (index = 0; index < fruits.length; index++) {  
    text += fruits[index];  
}
```

```
var person = [];  
person[0] = "John";  
person[1] = "Doe";  
person[2] = 46;  
var x = person.length;           // person.length will return 3  
var y = person[0];               // person[0] will return "John"
```

Array Object Creation

- Arrays can be created using the `new Array` method
 - `new Array` with one parameter creates an empty array of the specified number of elements

```
new Array(10);
```
 - `new Array` with no parameter creates an empty array

```
var a = new Array();  
a[0] = "dog"; a[1] = "cat"; a[2] = "hen";  
console.log(a.length); // outputs 3
```
 - `new Array` with two or more parameters creates an array with the specified parameters as elements

```
new Array(1, 2, "three", "four");
```
- Literal arrays can be specified using square brackets to include a list of elements

```
var alist = [1, "ii", "gamma", "4"];
```
- It is better to avoid the “new” keyword where possible
- Elements of an array *do not have to be of the same type*

Characteristics of Array Objects

- The length of an array is one more than the highest index
- You can iterate over an array using this length property, or you can use `for...in` construct

```
for(var i in a)
  console.log( a[i] );
```
- Assignment to an index greater than or equal to the current length simply increases the length of the array
 - `a[100] = "lion"; console.log(a.length);`
 - (Note: errors may go unnoticed.)
- Only assigned elements of an array occupy space
 - Suppose an array were created using `new Array(200)`
 - Suppose only elements 150 through 174 were assigned values
 - Only the 25 assigned elements would be allocated storage, the other 175 would not be allocated storage
- If you query a non-existent array index, you get undefined –

```
console.log(a[90])           // outputs undefined
```

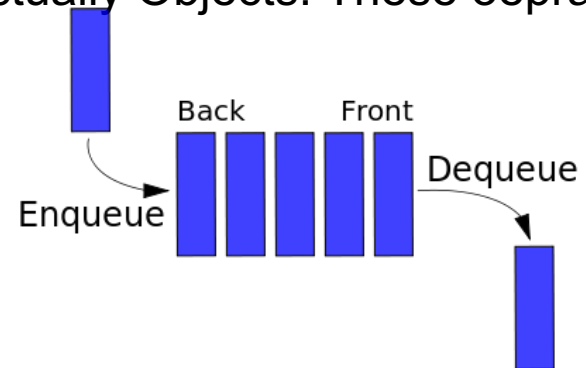
Array Methods

- `join` *returns a string of the elements in the array*
- `reverse` *....reverses the array*
- `sort` *.... sorts the array, can take a comparator function as an argument*
- `concat` *concatenates 2 or more arrays*
- `slice` *creates 2 arrays from 1 array*
- `splice` *inserts a group of elements at a given index*
- `delete` *replaces an element at an index with undefined*

Associative Arrays index on Strings and are actually Objects. These operations are not available to them:

```
var arr = [];  
arr["name"] = "Bob";
```

- `push`: Add to the end
- `pop`: Remove from the end
- `shift`: Remove from the front
- `unshift`: add to the front



Function Fundamentals



- Function definition syntax
 - A function definition consists of a header followed by a compound statement
 - A function header:
 - *function function-name(optional-formal-parameters)*
- Function call syntax
 - Function name followed by parentheses and any actual parameters
 - Function call may be used as an expression or part of an expression
- Functions must be defined before use in the page header (or linked in an external file)
- return statements
 - A return statement causes a function to cease execution and control to pass to the caller
 - A return statement may include a value which is sent back to the caller
 - If the function doesn't have any return statement, or uses an empty return with no value, then *undefined* is returned.

Functions

- Along with the objects, functions are the core components in understanding Javascript. We can also treat functions as objects. The most basic function is as follows

```
function add(x, y){  
    var total = x+y;  
    return total;  
}
```

- You can call the above function with no parameter as well. In such case, they will be set to *undefined*.

```
<body>
```

```
<h2>JavaScript Functions</h2>
```

```
<p>This example calls a function which performs a calculation, and  
returns the result:</p>
```

```
<p id="demo"></p>
```

```
<script>
```

```
function myFunction(p1, p2) {  
    return p1 * p2;  
}
```

```
document.getElementById("demo").innerHTML = myFunction(4, 3);
```

```
</script>
```

```
</body>
```

JavaScript Functions

This example calls a function which performs a calculation, and returns the result:

12

Functions are Objects

- *Functions are objects* in JavaScript (or *first class functions*)
- Functions may, therefore, be assigned to variables and to object properties
 - *Object properties that have function name as values are **methods** of the object*

Example

```
function fun() {  
    console.log("This surely is fun!");  
}  
ref_fun = fun; // Now, ref_fun refers to  
              // the fun object  
fun();         // A call to fun  
ref_fun();     // Also a call to fun
```

<h2>JavaScript Functions</h2>

<p>Accessing a function without () will return the function definition instead of the function result:</p>

<p id="demo"></p>

```
<script>  
function toCelsius(f) {  
    return (5/9) * (f-32);  
}  
document.getElementById("demo").innerHTML = toCelsius;  
</script>
```

JavaScript Functions

Accessing a function without () will return the function definition instead of the function result:

```
function toCelsius(f) { return (5/9) * (f-32); }
```

Local Variables

- “The *scope* of a variable is the range of statements over which it is visible”
- *A variable not declared using **var** has global scope*, visible throughout the page, even if used inside a function definition
- A variable declared with **var** outside a function definition has global scope
- A variable declared with **var** inside a function definition has local scope, visible only inside the function definition
 - If a global variable has the same name, it is hidden inside the function definition
- A variable declared with **let** or **const** has block level scope.

```
// code here can NOT use carName

function myFunction() {
  var carName = "Volvo";

  // code here CAN use carName
}
```

```
var carName = "Volvo";

// code here can use carName

function myFunction() {
  // code here can also use carName
}
```

```
myFunction();

// code here can use carName

function myFunction() {
  carName = "Volvo";
}
```

Parameters

- Parameters named in a function header are called ***formal parameters***
- Parameters used in a function call are called ***actual parameters***
- Use arguments to access non-formal parameters

```
x = findMax(1, 123, 500, 115, 44, 88);

function findMax() {
    var i;
    var max = -Infinity;
    for (i = 0; i < arguments.length; i++) {
        if (arguments[i] > max) {
            max = arguments[i];
        }
    }
    return max;
}
```

```
function fun1(my_list) {
    var list2 = new Array(1, 3, 5);
    my_list[3] = 14; //changes actual
    parameter
}
```

Parameters are *passed by value*

For an object parameter, the reference is passed, so the function body can actually change the object

However, an assignment to the formal parameter will not change the actual parameter

```
my_list = list2; //no effect on
actual parameter
```

```
return my_list;
}
```

```
var list = new Array(2, 4, 6, 8)
fun1(list);
```

Parameter Checking

- JavaScript checks neither the type nor number of parameters in a function call
 - Formal parameters have no type specified
 - Extra actual parameters are ignored (however, see below)
 - If there are fewer actual parameters than formal parameters, the extra formal parameters remain undefined
- This flexibility is typical of many scripting languages
 - different numbers of parameters may be appropriate for different uses of the function
- A property array named **arguments** holds all of the actual parameters, whether or not there are more of them than there are formal parameters

The `sort` Method

- A parameter can be passed to the `sort` method to specify how to sort elements in an array
 - The parameter is a function that takes two parameters
 - The function returns a negative value to indicate the first parameter should come before the second
 - The function returns a positive value to indicate the first parameter should come after the second
 - The function returns 0 to indicate the first parameter and the second parameter are equivalent as far as the ordering is concerned
- Example:

```
var points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return b>a});
```

Constructors

- Constructors are functions that create and initialize properties for new objects
- A constructor uses the keyword `this` in the body to reference the object being initialized
- Object methods are properties that refer to functions
 - A function to be used as a method may use the keyword `this` to refer to the object for which it is acting

```
function Person(first, last, age, eye) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eye;  
}
```

```
var person = {  
    firstName: "John",  
    lastName: "Doe",  
    age: 50,  
    eyeColor: "blue"  
};
```

```
var myFather = new Person("John", "Doe", 50, "blue");  
var myMother = new Person("Sally", "Rally", 48, "green");
```

Functions (Recursive)


- Like any other languages, you can write recursive functions in Javascript. However, this creates a problem if the function is anonymous. How would you call a function without its name? The solution is using *named anonymous functions* -

```
var ninja = {  
    yell: function cry(n) {  
        return n > 0 ? cry(n-1) + "a" : "hiy";  
    }  
};
```

```
console.log( ninja.yell(5) ); // outputs hiyaaaaa
```

Objects

- Javascript objects are simply collections of name-value pairs. As such, they are similar to HashMaps in Java. An object may be thought of as a Map/Dictionary/Associative-Storage.
- If a variable is not a primitive (undefined, null, boolean, number or string), its an object.
- The name part is a string, while the value can be any Javascript value – including more objects.

Object	Properties	Methods
	<code>car.name = Fiat</code> <code>car.model = 500</code> <code>car.weight = 850kg</code> <code>car.color = white</code>	<code>car.start()</code> <code>car.drive()</code> <code>car.brake()</code> <code>car.stop()</code>

Accessing Object Properties

- Just like Java, an object's properties can be accessed using the dot operator -
 - `Obj.name = "Tim French"`
- And using the array-like index –
 - `Obj["name"] = "Dr French";`
- Both of these methods are semantically equivalent.
- The second method has the advantage that the name of the property is provided as a string, which means it can be calculated at run-time. It can also be used to set and get properties with names that are reserved words.
- As functions are first class objects, you can also update methods at runtime.

```
person.name = function () {  
    return this.firstName + " " + this.lastName;  
};
```

Dynamic Properties

- Create my_car and add some properties

```
// Create an Object object
var my_car = new Object();
// Create and initialize the make property
my_car.make = "Ford";
// Create and initialize model
my_car.model = "Contour SVT";
```
- The delete operator can be used to delete a property from an object
 - delete my_car.model

for-in loop Syntax

for (identifier in object)

statement or compound statement

The loop lets the identifier take on each property in turn in the object

```
for (var prop in my_car)
  console.log("Key: ", prop, "; Value:", my_car[prop]);
```

Result:

Name: make; Value: Ford

Name: model; Value: Contour SVT

Object-orientation in Javascript



- Javascript doesn't have classes, so its object-oriented approach doesn't match that of other popular OOP languages like Java, C# etc. Instead, it supports a variation of Object-oriented programming known as **Prototype-based** Programming.
- In prototype-based programming, classes are not present, and behavior reuse (equivalent to *inheritance* in Java) is accomplished through a process of decorating existing objects which serves as prototypes. This model is also known as *class-less*, *prototype-oriented* or *instance-based programming*.
- Just like Java, every object in Javascript is an instance of the object *Object* and therefore inherits all its properties and methods.

The *this* keyword

- When used inside a function, *this* refers to the current object. What that actually means is specified by the way in which you called that function.
- In the global scope of a browser it refers to the window displaying the HTML.
- In Node, it refers to the execution environment.
- If you called it using the dot notation or bracket notation on an object, that object becomes *this*. Otherwise *this* refers to the global object (the window object). For example

```
s = makePerson("Simon", "Willison")
var fullName = s.fullName;
console.log( fullName() );
// will output undefined undefined
```

It has different values depending on where it is used:

In a method, `this` refers to the **owner object**.

Alone, `this` refers to the **global object**.

In a function, `this` refers to the **global object**.

In a function, in strict mode, `this` is `undefined`.

In an event, `this` refers to the **element** that received the event.

Methods like `call()`, and `apply()` can refer `this` to **any object**.

The *new* keyword

- **new** is strongly related to **this**. What it does is it creates a brand new empty object, and then calls the function specified, with this set to that new object. Functions that are designed to be called by new are called constructor functions.
- When the code **new Person(...)** is executed, the following things happen —
 1. A new object is created, inheriting from **Person.prototype**.
 2. The constructor function Person is called with the specified arguments and this bound to the newly created object. **new Person** is equivalent to **new Person ()**, i.e. if no argument list is specified, Person is called without arguments.
 3. The object returned by the constructor function becomes the result of the whole new expression. If the constructor function doesn't explicitly return an object, the object created in step 1 is used instead. (Normally constructors don't return a value, but they can choose to do so if they want to override the normal object creation process.)

Function objects reuse

- Every time we are creating a *person* object, we are creating two new brand new function objects within it. Wouldn't it be better if this code was shared? There are two ways in which code can be shared. The first way is the following

```
function personFullName() {  
    return this.first + ' ' + this.last;  
}  
function personFullNameReversed() {  
    return this.last + ', ' + this.first;  
}  
function Person(first, last) {  
    this.first = first;  
    this.last = last;  
    this.fullName = personFullName;  
    this.fullNameReversed =  
    personFullNameReversed;  
}
```

- The second (and best) way is to use the *prototype*

```
function Person(first, last){  
    this.first = first;  
    this.last = last;  
}  
Person.prototype.fullName = function() {  
    return this.first + ' ' + this.last;  
}  
Person.prototype.fullNameReversed =  
    function() {  
        return this.last + ', ' + this.first;  
    }
```

The *prototype*

- *Person.prototype* is an object shared by all instances of Person. It forms a part of a lookup chain (or, *prototype chain*) : any time you attempt to access a property of *Person* that isn't set, Javascript will check *Person.prototype* to see if that property exists there instead. As a result, anything assigned to *Person.prototype* becomes available to all instances of that constructor via the *this* object. The root of the prototype chain is *Object.prototype*.
- This is an incredibly powerful tool. Javascript lets you modify something's prototype at anytime in your program, which means you can add extra methods to existing objects at runtime.

Adding methods at run time using prototype

```
var s = "Issa";  
String.prototype.reversed = function(){  
    var r = "";  
    for (var i = this.length - 1; i >= 0; i--)  
    {  
        r += this[i];  
    }  
    return r;  
}  
s.reversed();           // will output assi
```

```
function Person(first, last, age, eyecolor) {  
    this.firstName = first;  
    this.lastName = last;  
    this.age = age;  
    this.eyeColor = eyecolor;  
}  
  
Person.prototype.nationality = "English";
```

```
"This can now be reversed".reversed()  
// outputs desrever eb won nac sihT
```

Javascript can also use prototypes to implement inheritance. A subclass can be defined to have the prototype of a superclass, and then the implementation of the methods can be overwritten in the subclass prototype..

Inner functions

- JavaScript function declarations are allowed inside other functions

```
function Example() {  
    var a = 1;  
    function oneMoreThanA() {  
        return a + 1;  
    }  
    return oneMoreThanA();  
}
```

What does `Example()` return?

- A closure is the local variables for a function – kept alive after the function has returned.

- Using inner functions we can use one of the most powerful abstractions Javascript has to offer – closure. A quick quiz, what does this do –

```
function makeAdder(a) {  
    return function(b) {  
        return a + b;  
    }  
}
```

```
x = makeAdder(5);  
y = makeAdder(20);
```

```
console.log( x(6) );  
// ?  
console.log( y(7) );  
// ?
```

Javascript closure

- Here, the outer function (makeAdder) has returned, and hence common sense would seem to dictate that its local variable no longer exist. But they do still exist, otherwise the adder function would be unable to work.
- In actuality, whenever Javascript executes a function, a scope object is created to hold the local variables created within that function. It is initialized with any variables passed in as function parameters.
- This is similar to the global object that all global variables and functions live in, but with a couple of important differences: firstly, a brand new scope object is created every time a function starts executing, and secondly, unlike the global object these scope objects cannot be directly accessed from your code.
- So when makeAdder is called, a scope object is created with one property: a, which is the argument passed to the function. It then returns a newly created function.
- Normally JavaScript's garbage collector would clean up the scope object created for makeAdder at this point, but the returned function maintains a reference back to that scope object. As a result, the scope object will not be garbage collected until there are no more references to the function object that makeAdder returned.

Javascript IO

- Standard output for JavaScript embedded in a browser is the window displaying the page in which the JavaScript is embedded
- Writing to the document object is now considered bad practice. For simple debugging use

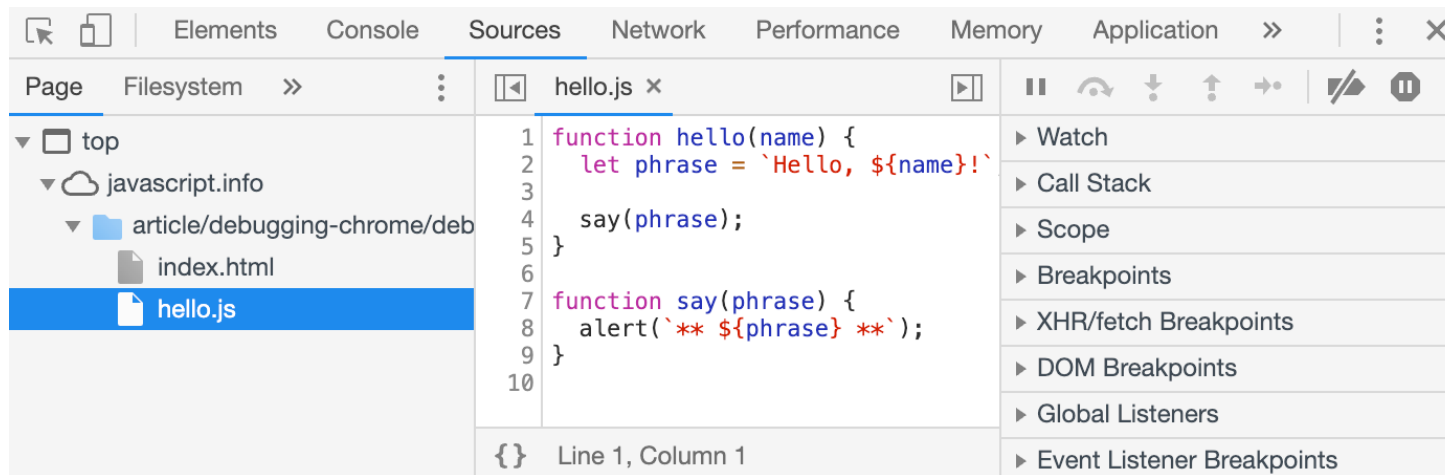
```
console.log("The result is: ", result, "<br />");
```

- To read, you can use *alert* or *confirm*. To get input you can use *prompt*.
- In NodeJS you can access `stdin`, `stdout`, and `stderr` through the *process* object. Eg:

```
const readline = require('readline');
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout});
rl.question('What do you think of Node.js? ', (answer) => {
  console.log('Thank you for your feedback:', answer);
  rl.close();});
```

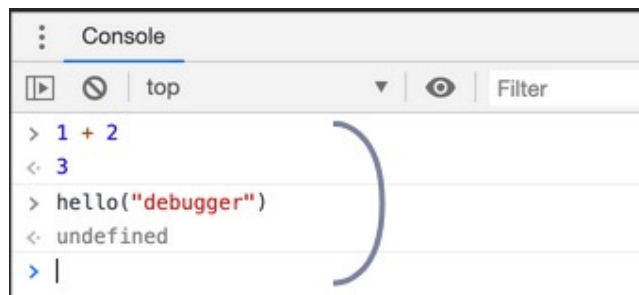
Debugging in the Browser

- JavaScript errors are detected by the browser. We will look at Chrome, but other browsers offer similar functionality:



• DevTools allows you to browse code.

• Console allows you to interact with the code and the page.



• Breakpoints and debugger; command allow you to interact with the code as it executes.