Table of Contents

[Objects 3](#_Toc452242218)

[Creation of objects 3](#_Toc452242219)

[Object inheritance 4](#_Toc452242220)

[Property Access Errors 4](#_Toc452242221)

[Deleting Properties 5](#_Toc452242222)

[Enumerating Properties 5](#_Toc452242223)

[Property Getters and Setters 5](#_Toc452242224)

[Property Attributes 6](#_Toc452242225)

[Object Attributes 7](#_Toc452242226)

[The prototype attribute 7](#_Toc452242227)

[The class attribute 8](#_Toc452242228)

[The Extensible Attribute 8](#_Toc452242229)

[Serializing Objects 9](#_Toc452242230)

[Object Methods 9](#_Toc452242231)

[The toString method 9](#_Toc452242232)

[The toJSON() method 10](#_Toc452242233)

[The valueOf() method 10](#_Toc452242234)

[Functions 10](#_Toc452242235)

[Defining Functions 10](#_Toc452242236)

[Constructor Invocation 12](#_Toc452242237)

[Indirect invocation 12](#_Toc452242238)

[Function Arguments and Parameters 12](#_Toc452242239)

[Optional Parameters 12](#_Toc452242240)

[Variable length argument lists: The Argument object 12](#_Toc452242241)

[Functions as Namespaces 13](#_Toc452242242)

[Variable Scope 13](#_Toc452242243)

[Function Scope and Hoisting 14](#_Toc452242244)

[Variables as Properties 15](#_Toc452242245)

[The Scope Chain 15](#_Toc452242246)

[Closures 15](#_Toc452242247)

[Function Properties, Methods 17](#_Toc452242248)

[The length property 17](#_Toc452242249)

[The prototype property 17](#_Toc452242250)

[The call() and apply() methods 17](#_Toc452242251)

[The bind() method 18](#_Toc452242252)

[The toString() Method 18](#_Toc452242253)

[Classes and Constructors 19](#_Toc452242254)

[Constructors and Class Identity 20](#_Toc452242255)

[Java style classes in JavaScript 20](#_Toc452242256)

[Constructor Object 20](#_Toc452242257)

[Prototype object 21](#_Toc452242258)

[Instance object 21](#_Toc452242259)

[Class definition in JavaScript 21](#_Toc452242260)

[Defining a SubClass 21](#_Toc452242261)

[Constructor and Method Chaining 23](#_Toc452242262)

[Preventing Class Extensions 23](#_Toc452242263)

[Execution of JavaScript Programs 24](#_Toc452242264)

[Security 25](#_Toc452242265)

[The Same Origin Policy 25](#_Toc452242266)

[Document Elements as Window Properties 26](#_Toc452242267)

[Multiple Windows and Frames 27](#_Toc452242268)

[Opening and Closing Windows 27](#_Toc452242269)

[Window Names 28](#_Toc452242270)

[Relationships between Frames 29](#_Toc452242271)

[JavaScript in Interacting Windows 30](#_Toc452242272)

[Selecting Document Elements 31](#_Toc452242273)

[Selecting Elements by Name 32](#_Toc452242274)

[Selecting Elements by Type 32](#_Toc452242275)

[NodeLists and HTMLCollections 33](#_Toc452242276)

[Selecting Elements by CSS class 33](#_Toc452242277)

[Selecting Elements with CSS Selectors 34](#_Toc452242278)

[Document Structure and Traversal 34](#_Toc452242279)

[Documents as Trees of Nodes 34](#_Toc452242280)

[Documents as Trees of Elements 35](#_Toc452242281)

[HTML Attributes as Element properties 35](#_Toc452242282)

[Getting and Setting non HTML attributes 36](#_Toc452242283)

[Dataset Attributes 36](#_Toc452242284)

[Attributes as Attr Nodes 36](#_Toc452242285)

[Element Content 37](#_Toc452242286)

[Element Content as HTML 37](#_Toc452242287)

[Element Content as Plain Text 38](#_Toc452242288)

[Element Content as Text Nodes 38](#_Toc452242289)

[Creating, Inserting and Deleting Nodes 38](#_Toc452242290)

[Creating Nodes 39](#_Toc452242291)

[Inserting Nodes 39](#_Toc452242292)

[Removing and Replacing Nodes 39](#_Toc452242293)

[Using DocumentFragments 40](#_Toc452242294)

[HTML Forms 40](#_Toc452242295)

[Selecting Forms and Form Elements 40](#_Toc452242296)

[Form and Element Properties 41](#_Toc452242297)

[Form and element Event Handlers 42](#_Toc452242298)

[Overview of CSS 42](#_Toc452242299)

[Ways of associating CSS styles with HTML Elements 43](#_Toc452242300)

[Inline style 43](#_Toc452242301)

[Using StyleSheets 43](#_Toc452242302)

[The Cascade 43](#_Toc452242303)

[Shortcut properties 44](#_Toc452242304)

[Non-standard properties 44](#_Toc452242305)

[Positioning Elements with CSS 44](#_Toc452242306)

[static 45](#_Toc452242307)

[absolute 45](#_Toc452242308)

[fixed 45](#_Toc452242309)

[relative 45](#_Toc452242310)

[The third dimension: z-index 46](#_Toc452242311)

[Borders, Margins and Padding 46](#_Toc452242312)

[The CSS Box Model and Positioning Details 47](#_Toc452242313)

[The border-box model and the box-sizing property 49](#_Toc452242314)

[Element Display and Visibility 49](#_Toc452242315)

Core Javascript Notes

# Objects

Any value in JavaScript that is not a string, a number, true, false, null, or undefined is an object.

# Creation of objects

Objects can be created using the literal syntax i.e. {}, with new and in ECMA5 with Object.create()

When creating an object with the {} syntax or with Object.create() its prototype will be set to Object.prototype

The new operator creates and initializes a new Object. The new keyword must be followed by a function invocation. A function used in this way is called a constructor and serves to initialize a newly created object.

Object.create() is a static function and creates a new object using its first argument as the prototype of that object. Object.create() also takes an optional second argument that describes the properties of the new object.

# Object inheritance

Javascript objects have a set of own properties and they also inherit a set of properties from their prototype object. Now suppose you assign to the property x of the object o. If o already has an own noninherited property named x, then the assignment simple changes the

value of this existing property. Otherwise, the assignment creates a new property named x on the object o. If o previously inherited the property x, that inherited property is now hidden by the newly created own property with the same name.

Property assignment examines the prototype chain to determine whether the assignment is allowed. If o inherits a read-only property named x, for example, then the assignment is not allowed. If the assignment is allowed, however, it always creates or sets a property in the

original object and never modifies the prototype chain. There is one exception to this rule and that is if o inherits the property x, and that property is an accessor property with a setter method, then that setter method is called rather than creating a new property x in o.

Note, however that the setter method is called on the object o not on the prototype object that defines the property, so if the setter method defines any properties, it will do so on o, and it will again leave the prototype chain unmodified.

The fact that inheritance occurs when querying properties but not when setting them is a key feature of JavaScript because it allows us to selectively override inherited properties.

# Property Access Errors

It is not an error to query a property that does not exist. If the property x is not found as an own property or an inherited property of o, the property access expression o.x evaluates to undefined. The null and undefined values have no properties and it is an error to query

properties of these values.

e.g var o ={ a:"propertyA" }

console.log(o.a.length); //returns length of the string defined by a

console.log(o.b); //returns undefined

console.log(o.b.length); //throws TypeError because undefined has no properties

# Deleting Properties

The delete operator removes a property from an object. The delete operator only deletes own properties, not inherited ones. To delete an inherited property, you must delete it from the prototype object in which it is defined but doing so will affect every object that inherits

from this object.

delete does not remove properties that have a configurable attribute of false though it will remove configurable properties of nonextensible objects. Certain properties of built in objects are nonconfigurable as are properties of the global object created by variable declaration

and function declaration.

# Enumerating Properties

The for/in loop runs the body of the loop once for each enumerable property (own or inherited) of the specified object, assigning the name of the property to the loop variable. Built in methods that objects inherit are not enumerable, but properties that your code adds to

objects are enumerable unless you make them non enumerable.

Prior to ECMA5, there is no way to make these added methods non-enumerable, so they are enumerated by for/in loops.

Method hasOwnProperty of an object tests whether that object has an own property with the given name. It returns false for inherited properties.

Method propertyIsEnumerable returns true only if the named property is an own property and is its enumerable attribute is set to true.

# Property Getters and Setters

An object property is a name and a value and a set of attributes.

In ECMA5 the value of a property may be replaced by one or two methods known as a getter and a setter. Properties defined by getters and setters are sometimes known as accessor properties to distinguish them from data properties that have a simple value.

When a program queries the value of an accessor property the getter method is invoked and when a program sets the value of an accessor property, the setter method is invoked. Accessor properties do not have a writable attribute as data properties do. if a property has

both a getter and a setter method, it is a read/write property. If it has only a getter method, it is a read only property and if it has only a setter method it is write-only something that is not possible with data properties.

Accessor properties are defined as one or two functions whose name is the same as the property name, and with the function keyword replaced with get and/or set.

e.g.

var o = {

data\_prop: value, //An ordinary data property

//accessor\_prop is an accessor property

get accessor\_prop(){},

set accessor\_prop(){}

}

# Property Attributes

In addition to a name and value, properties have attributes that specify whether they can be written, enumerated and configured. The ECMA5 methods for querying and setting the attributes of a property use an object called a property descriptor to represent the set of

four attributes.

The property descriptor object for a data property has properties named value, writable, enumerable and configurable.

The property descriptor object for an accessor property has properties named get, set, enumerable and configurable.

To obtain the property descriptor for a named property of a specified object, call Object.getOwnPropertyDescriptor; returns undefined for inherited properties and properties that don't exist. To query the attributes of inherited properties, you must explicitly traverse the

prototype chain.

To set the attributes of a property, or to create a new property with the specified attributes, call Object.defineProperty(), passing the object to be modified, the name of the property to be created or altered, and the property descriptor object:

e.g.

var o = {};

Object.defineProperty(o,"x", {

value:1,

writable:true,

enumberable:true,

configurable:true

});

If you want to create or modify more than one property at a time, use Object.defineProperties()

e.g.

var p = Object.defineProperties({}, {

x: {value:1, writable:true, enumerable:true, configurable:true},

r: {get: function(){/\*get function\*/}, enumerable:true, configurable:true}

});

Object.defineProperty and Object.defineProperties will throw TypeError if the attempt to create or modify a property is not allowed. For a list of rules look at Page 133 of the Javascript Definitive Guide 6th edition book.

# Object Attributes

Every object has associated prototype, class and extensible attributes.

# The prototype attribute

An object's prototype attribute specifies the object from which it inherits properties. The prototype attribute is set when an object is created. Objects created from object literals use Object.prototype as their prototype. Objects created with new use the value of the

prototype property of their constructor function as their prototype and objects created with Object.create() use the first argument to that function (which may be null) as their prototype. If the first argument is null then they use Object.prototype as their prototype.

In ECMA5 you can query the prototype of any object by passing the object to Object.getPrototypeOf(). There is no equivalent in ECMA3 but it is often possible to determine the prototype of an object o using the expression o.constructor.prototype. Objects created with

new expression usually inherit a constructor property that refers to the constructor function used to create the object and as mentioned constructor functions have a prototype property that specifies the prototype for objects created using that constructor. Objects

created by object literals or by Object.create() have a constructor property that refers to the Object() constructor. Thus, constructor.prototype refers to the correct prototype for object literals, but does not usually do so for objects created with Object.create.

To determine if one object is the prototype of or is part of the prototype chain of another object, use the isPrototypeOf() method.

## The class attribute

An object's class attribute is a string that provides information about the type of the object. Neither ECMA3 or ECMA5 provide any way to set this attribute and there is only an indirect technique for querying it. The default toString() method inherited from Object.prototype returns a string of this form:

[object class]

So to obtain the class of an object you can invoke this toString() method on it, unless it has been replaced by some other inherited toString() method. Therefore this attribute can be queried indirectly using Object.prototype.toString.call() method.

Objects created through built in constructors such as Array and Date have class attributes that match the names of their constructors. Host objects typically have meaningful class attributes as well, though this is implementation dependent. Objects created through

object literals or by Object.create have a class attribute of Object.

## The Extensible Attribute

The extensible attribute of an object specifies whether new properties can be added to the object or not. In ECMA3, all built in and user-defined objects are implicitly extensible, and the extensibility of host objects is implementation defined. In ECMA5 all built in and

user defined objects are extensible unless they have been converted to be nonextensible and again the extensibility of host objects is implementation defined.

To determine whether an object is extensible, pass it to Object.isExtensible(). To make an object nonextensible, pass it to Object.preventExtensions(). Note that there is no way to make an object extensible again once you have made it nonextensible. Also note that

calling preventExtensions() only affects the extensibility of the object itself. If new properties are added to the prototype of a nonextensible object, the nonextensible object will inherit those new properties.

The purpose of the extensible attribute is to be able to lock down objects into a known state and prevent outside tampering. The extensible object attribute is often used in conjunction with the configurable and writable property attributes, and ECMA5 defines functions

that make it easy to set these attributes together.

Object.seal() works like Object.preventExtensions(), but in addition to making the object non-extensible, it also makes all of the own properties of that object non-configurable. This means that new properties cannot be added to the object, and existing properties cannot be

deleted or configured. Existing properties that are writable can still be set, however. There is no way to unseal a sealed object. You can use the Object.isSealed() to determine whether an object is sealed.

Object.freeze() locks objects down even more tightly. In addition to making the object non-extensible and its properties non-configurable, it also makes all of the object's own data properties read-only. (If the object has accessor properties with setter methods these are not

affected and can still be invoked by assignment to the property.) Use Object.isFrozen to determine if the object is frozen.

It is important to understand that Object.seal and Object.freeze all return the object that they are passed: they have no effect on the prototype of that object. If you want to thoroughly lock down an object, you probably need to seal or freeze the objects in the prototype

chain as well. They also return the object that they are passed which means you can use them in nested function invocations.

## Serializing Objects

Object serialization is the process of converting an object's state to a string from which it can later be restored. ECMA5 provides native function JSON.stringify() and JSON.parse() to serialize and restore JavaScript objects.

# Object Methods

## The toString method

The toString() method takes no arguments and returns a string that somehow represents the value of the object on which it is invoked. The default toString() method is not very informative though it is useful for determining the class of an object. Because this default

method does not display much useful information, many classes define their own versions of toString(). For example when an array is converted to a string, you obtain a list of the array elements, themselves each converted to a string, and when a function is converted

to a string you obtain the source code for the function.

## The toJSON() method

Object.prototype does not actually define a toJSON() method, but the JSON.stringify() method looks for a toJSON() method on any object it is asked to serialize. If this method exists on the object to be serialized, it is invoked and the return value is serialized instead of the

original object.

## The valueOf() method

The valueOf method is much like the toString() method, but it is called when JavaScript needs to convert an object to some primitive type other than a string - typically a number. Javascript calls this method if an object is used in a context where a primitive value is required.

# Functions

## Defining Functions

Functions are defined with the function keyword which can be used in a function definition expression or in a function declaration statement. See section 5.3.2 for more.

e.g. function definition

var f = function(){/\*\*/}

function declaration

function f(){/\*\*/}

Function declaration statements may appear in top level JavaScript code or they may be nested within other functions. When nested, however, function declarations may only appear at the top level of the function they are nested within. That is function definitions may not appear within if statements, while loops or any other statements.( I would take this last sentence with a bit of skepticism because it seems to work in current browsers).

Function declaration statements differ from function definition expressions in that they include a function name. Both forms create a new function object, but the function declaration statement also declares the function name as a variable and assigns the function object to it. Like variables declared with var, functions defined with function definition statements are implicitly hoisted to the top of the containing script or function, so that they are visible throughout the script or function. With var, only the variable declaration is hoisted-the variable initialization code remains where you placed it. With function declaration statements, however, both the function name and the function body are hoisted: all functions in a script or all nested functions in a function are declared before any other code is run. This means that you can invoke a JavaScript function before you declare it.

Like the var statement, function declaration statements create variables that cannot be deleted. These variables are not read-only however, and their value can be overwritten.

Unlike variables, the this keyword does not have a scope, and nested functions do not inherit the this value of the containing function. If a nested function is invoked as a method, it’s this value is the object it was invoked on. If a nested function is invoked as a function, then it’s this value will be either the global object (non-strict mode) or undefined (strict mode). It is a common mistake to assume that a nested function invoked as a function can use this to obtain the invocation context of the outer function. If you want to access the this value of the outer function, you need to store that value into a variable that is in scope for the inner function. It is common to use the variable self for this purpose.

e.g.

var o = {

m: function(){

var self = this;

console.log(this === o) //prints true

f();

}

function f(){

console.log(this === o); //false: this is global or undefined

console.log(self === o); //true: self is outer this value

}

};

o.m();

# Constructor Invocation

A constructor invocation creates a new, empty object that inherits from the prototype property of the constructor. Constructor functions are intended to initialize objects and this newly created object is used as the invocation context, so the constructor function can refer to it with the this keyword. Note that the new object is used as the invocation context even if the constructor invocation looks like a method invocation. That is, in the expression new o.m(), o is not used as the invocation context.

Constructor invocations do not use the return keyword. They typically initialize the new object then return implicitly the new object that was created. If a constructor explicitly used the return statement to return an object, then that object becomes the value of the invocation expression. If the constructor uses return with no value or if it returns a primitive value, that return value is ignored and the new object is used as the value of the invocation.

# Indirect invocation

JavaScript functions are objects and like all JavaScript objects, they have methods. Two of these methods, call() and apply(), invoke the function indirectly. Both methods allow you to explicitly specify the this value for the invocation, which means you can invoke any function as a method of any object, even if it is not actually a method of that object. Both methods allow you to specify the arguments for the invocation. The call() method uses its own argument list as arguments to the function and the apply() method expects an array of values to be used as arguments.

# Function Arguments and Parameters

## Optional Parameters

When a function is invoked with fewer arguments than declared parameters, the additional parameters are set to the undefined value.

# Variable length argument lists: The Argument object

When a function is invoked with more argument values than there are parameter names, there is no way to directly refer to the unnamed values. The Arguments object provides a solution to this problem. Within the body of a function, the identifier arguments refer to the Arguments object for that invocation. The Arguments object is an array like object that allows the argument values passed to the function to be retrieved by number, rather than by name.

Each Argument object defines numbered array elements and a length property, but it is not technically an array.

The Arguments object has one very unusual feature in non-strict mode, when a function has named parameters, the array elements of the Arguments object are aliases for the parameters that hold the function arguments. The numbered elements of the Arguments object and the parameter names are like two different names for the same variable. Changing the value of an argument with an argument name changes the value that is retrieved through the arguments[] array. Conversely, changing the value of an argument through the arguments[] array changes the value that is retrieved by the argument name.

This special behavior of Arguments object has been removed in the strict mode of ECMA5.

# Functions as Namespaces

JavaScript has function scope: variables declared within a function are visible throughout the function including within nested functions but do not exist outside of the function. Variables declared outside of a function are global variables and are visible throughout your JavaScript program. JavaScript does not define any way to declare variables that are hidden within a single block of code and for this reason, it is sometimes useful to define a function simply to act as a temporary namespace in which you can define variables without polluting the global namespace.

e.g.

function mymodule(){

//Module code goes here

//Any variables used by the module are local to this function

//instead of cluttering up the global namespace

}

mymodule(); //But don’t forget to invoke the function!

Alternatively you can define and invoke an anonymous function in a single expression

(function(){

//Module code goes here

}()); //end the function literal and invoke it now.

The open parenthesis before function is required because without it, the JavaScript interpreter tries to parse the function keyword as a function declaration statement. With the parenthesis, the interpreter correctly recognizes this as a function definition expression.

# Variable Scope

The scope of a variable is the region of your program in which it is defined. A global variable has global scope; it is defined everywhere in your JavaScript code. On the other hand, variables declared within a function are defined only within the body of the function. They are local variables and have local scope. Function parameters also count as local variables and are defined only within the body of the function.

Within the body of a function, a local variable takes precedence over a global variable with the same name.

# Function Scope and Hoisting

In some C like programming languages, each block of code within curly braces has its own scope, and variables are not visible outside of the block in which they are declared. This is called block scope, and JavaScript does not have it. Instead, JavaScript uses function scope: variables are visible within the function in which they are defined and within any functions that are nested within that function.

JavaScript’s function scope means that all variables declared within a function are visible throughout the body of the function. JavaScript code behaves as if all the variable declarations in a function are hoisted to the top of the function. Even though the local variable is defined throughout the function, it is not actually initialized until the var statement is executed.

e.g.

function f(){

console.log(scope); //variable scope is defined so it prints undefined here

var scope = ‘local’; //variable scope is initialized here

console.log(scope); //prints local here

console.log(‘End’);

}

The above function is equivalent to:

function f(){

var scope;

console.log(scope); //variable scope is defined so it prints undefined here

scope = ‘local’; //variable scope is initialized here

console.log(scope); //prints local here

console.log(‘End’);

}

The variable is hoisted to the top.

# Variables as Properties

When you declare a global JavaScript variable, what you are actually doing is defining a property of the global object. If you use var to declare the variable, the property that is created is nonconfigurable, which means that it cannot be deleted with the delete operator. If you are not using strict mode and you assign a value to an undeclared variable, JavaScript automatically creates a global variable for you. Variables created in this way are regular configurable properties of the global object and they can be deleted.

# The Scope Chain

JavaScript is a lexically scoped language: the scope of a variable can be thought of as the set of source code lines for which the variable is defined. Global variables are defined throughout the program. Local variables are defined throughout the function in which they are declared, and also within any functions nested within that function.

When JavaScript needs to look up the value of a variable x (a process called variable resolution), it starts by looking at the first object in the chain. If that object has a property named x, the value of that property is used. If the first object does not have a property named x, JavaScript continues the search with the next object in the chain. If the second object does not have a property named x, the search moves on to the next object and so on. If x is not a property of any of the objects in the scope chain, then x is not in scope for that code, and a ReferenceError occurs.

In top-level JavaScript code, the scope chain consists of a single object, the global object. In a non-nested function, the scope chain consists of two objects. The first is the object that defines the function’s parameters and local variables, and the second is the global object. In a nested function, the scope chain has three or more objects. It is important to understand how this chain of objects is created.

When a function is defined, it stores the scope chain then in effect. When that function is invoked, it creates a new object to store its local variables and adds that new object to the stored scope chain to create a new, longer chain that represents the scope for that function invocation. This becomes more interesting for nested functions because each time the outer function is called, the inner function is defined again.

# Closures

JavaScript uses lexical scoping. This means that functions are executed using the variable scope that was in effect when they were defined, not when the variable scope that is in effect when they are invoked.

e.g.

var scope = “global scope”;

function checkscope(){

var scope = “local scope”;

function f(){ return scope; }

return f();

}

checkscope(); // local scope is returned

The above function is modified slightly to return a function instead of the return value for that function.

var scope = “global scope”;

function checkscope(){

var scope = “local scope”;

function f(){ return scope; }

return f;

}

checkscope()(); // local scope is still returned!

Remember the fundamental rule of lexical scoping: JavaScript functions are executed using the scope chain that was in effect when they were defined. The nested function f() was defined under a scope chain in which the variable scope was bound to the value “local scope”. That binding is still in effect when f is executed, wherever it is executed from. So the last line of code above returns “local scope”, and not “global scope”

This is in essence how you can create private variables in JavaScript. Private variables created in this way need not be exclusive to a single closure. It is perfectly possible for two or more nested functions to be defined within the same outer function and share the same scope chain.

e.g.

function counter(){

var n = 0;

return {

count: function() { return n++ },

reset: function() { n = 0; }

};

}

var c = counter(), d = counter(); //creates two counters

c.count(); // => 0

d.count(); // => 0 they count independently

c.reset(); // reset() and count methods share state

c.count() // => 0; because c was reset

d.count(); // => 1; d was not reset

The first thing to understand is that the two methods share access to the private variable n (because javascript has function scope, n is visible throughout the function). The second thing to understand is that each invocation of counter() creates a new scope chain (because count and reset functions get defined when counter() is invoked) and a new private variable.

# Function Properties, Methods

Functions are really a specialized kind of JavaScript object. Since functions are objects, they can have properties and methods like any other object. There is even a Function() constructor to create new function objects.

## The length property

Within the body of a function arguments.length specifies the number of arguments that were passed to the function. The length property of a function itself, however, has a different meaning. This read-only property returns the arity of the function-the number of parameters it declares in it’s parameter list, which is usually the number of arguments that the function expects.

## The prototype property

Every function has a prototype property that refers to an object known as the prototype object. Every function has a different prototype object. When a function is used as a constructor, the newly created object inherits properties from the prototype object.

## The call() and apply() methods

Call() and apply() allow you to indirectly invoke a function as if it were a method of some other object. The first argument to both call() and apply() is the object on which the function is to be invoked; this argument is the invocation context and becomes the value of the this keyword within the body of the function.

In ECMA5 strict mode, the first argument to call() or apply() becomes the value of this, even if it is a primitive value or null or undefined. In ECMA3 and non strict mode, a value of null or undefined is replaced with the global object and a primitive value is replaced with the corresponding wrapper object.

Any arguments to call() after the first invocation context argument are the values that are passed to the function that is invoked.

The apply() method is like the call() method, except that the arguments to be passed to the function are specified as an array.

f.apply(o,[1,2]);

If a function is defined to accept an arbitrary number of arguments, the apply() method allows you to invoke that function on the contents of an array of arbitrary length. Note the apply() method works with array-like objects as well as true arrays.

## The bind() method

The bind method was added in ECMA5 and when you invoke the method on a function f and pass an object o, the method returns a new function. Invoking the new function invokes the original function f as a method of o. Any arguments that you pass to the new function are passed to the original function.

The bind function does more than just bind a function to an object. It also performs partial application: any arguments you pass to bind() after the first are bound along with the this value.

e.g.

var sum = function(x,y){ return x + y }; //return the sum of 2 args

//Create a new function like sum, but with the this value bound to null and the 1st argument

//bound to 1. This new function expects just one argument

var succ = sum.bind(null, 1);

succ(2); // => 3; x is bound to 1 and we pass 2 for the y argument

function f(y,z){ return this.x + y + z };

var g = f.bind({x:1},2);

g(3); // => 6; this.x is bound to 1, y is bound to 2 and z is 3

## The toString() Method

Like all JavaScript objects, functions have a toString() method. The ECMA spec requires this method to return a string that follows the syntax of the function declaration statement. In practice most (but not all) implementations of this toString() method return the complete source code for the function. Built in functions typically return a string that includes something like “[native code]” as the function body.

# Classes and Constructors

Constructors are invoked using the new keyword. Constructor invocations using new automatically create a new object and invoke the constructor as a method of that object and return the new object. The critical feature of constructor invocations is that the prototype property of the constructor is used as the prototype of the new object.

Any JavaScript function can be used as a constructor, and constructor invocations need a prototype property. Therefore, every JavaScript function (except functions returned by the ECMA5 bind method) automatically has a prototype property. The value of this property is an object that has a single nonenumerable constructor property. The value of the constructor property is the function object.

i.e.

var f = function(){

}

* f:{

protoype:{

constructor:f

}

e.g. Creating a Range class of objects

function Range(from to){

//Perform initialization or properties that are unique to the new object being created

this.from = from;

this.to = to;

}

//Define the prototype object for the constructor function

Range.prototype = {

constructor: Range, //explicitly set the constructor back reference

includes: function(x){ /\*\*/}

foreach:function(f){ /\*\*/ }

toString: function() { /\*\*/ }

}

Create a new class object

var r = new Range(1,3);

r.includes(2);

r.foreach(console.log);

console.log(r);

Note: we could have predefined the constructor object as follows as well:

Range.prototype.includes = function(x) { /\*\*/ };

Range.prototype.foreach = function(f) { /\*\*/ };

Range.prototype.toString = function() {/\*\*/};

We don’t have to do anything to define the constructor property because Functions already have a prototype object with a constructor property in it.

## Constructors and Class Identity

Two objects are instances of the same class if and only if they inherit from the same prototype object. Constructors are used with the instanceof operator when testing objects for membership in a class. If we have an object r and want to know if it is a Range object we can write:

r instanceof Range //returns true if r inherits from Range.prototype

The instanceof operator does not actually check whether r was initialized by the Range constructor. It checks whether it inherits from Range.prototype. The inheritance need not be direct. If r inherits from an object that inherits from an object that inherits from Range.prototype, the expression will still evaluate to true. (9.5.1).

A word of caution when using instanceof operator in client side JavaScript that uses more than one window or frame is that each window or frame is a distinct execution context, and each has its own global object, and its own set of constructor functions. Two arrays created in two different frames inherit from two identical but distinct prototype objects, and an array created in one frame is not instanceof the Array() constructor of another frame.

## Java style classes in JavaScript

In JavaScript, there are three different objects involved in any class definition, and the properties of these three objects act like different kinds of class members:

### Constructor Object

The constructor function defines a name for a JavaScript class. Properties you add to this constructor object serve as class fields and class methods.

### Prototype object

The properties of this object are inherited by all instances of the class and properties whose values are functions behave like instance methods of the class.

### Instance object

Each instance of a class is an object in its own right, and properties defined directly on an instance are not shared by any other instances. Non function properties defined on instances behave as the instance fields of the class.

## Class definition in JavaScript

The process of class definition in JavaScript can be reduced to a three step algorithm. First write the constructor function that sets instance properties on new objects. Second, define instance methods on the prototype object of the constructor. Third, define class fields and class properties on the constructor itself.

# Defining a SubClass

JavaScript objects inherit properties (usually methods) from the prototype object of their class.

If an Object is an instance of a class B and B is a subclass of A, then Object must also inherit properties from A. We arrange this by ensuring that the prototype object of B inherits from the prototype object of A.

B.prototype = inherit(A.prototype); //subclass inherits from superclass

B.prototype.constructor = B; //Override the inherited constructor prop

These two lines of code are the key to creating subclasses in JavaScript. Without them, the prototype object will be an ordinary object, an object that inherits from Object.prototype and this means that your class will be a subclass of Object like all classes are. *(that is why we don’t simply assign an object literal to the prototype property. We create the object using the object literal to base it as its prototype then use Object.create or assign the object literal to the prototype property of a constructor function and then use the constructor to create a new object which is then assigned to the subclass prototype.*

e.g. subclass definition function

function defineSubclass( superclass, //Constructor of the superclass

constructor, //The constructor for the new subclass

methods, //Instance methods: copied to the prototype

statics, //Class properties: copied to the constructor

){

//Set up the prototype of the subclass

constructor.prototype = inherit(superclass.prototype);

constructor.prototype.constructor = constructor;

//copy the methods and statics as we would do for a regular class

if(methods) extend(constructor.prototype, methods);

if(statics) extend(constructor, statics);

//return the class

return constructor;

}

//We can also do this as a method of the superclass constructor

Function.prototype.extend = function(constructor, methods, statics){

Return defineSubclass(this, constructor, methods, statics);

};

Note: we can also use Object.create to extend a class. See example 9-18 for more

//Extend is defined as a utility function that is added to Object (section //6.5)

//Copy the enumerable properties of p to o, and return o.

function extend(o,p){

for(prop in p){

o[prop] = p[prop];

}

Return o;

}

//inherit function (Example 6.1)

//inherit() returns a newly created object that inherits properties from the //prototype object p. It uses the ECMAScript 5 function object.create() if it

//is not defined, and otherwise falls back to an older technique

function inherit(p){

if( p == null ) throw TypeError(); //p must be a non-null object

if(Object.create)

return Object.create(p); //if Object.create is defined just use //it.

var t = typeof p;

if( t!== “object” && t !== “function” ) throw TypeError();

function f(){};

f.prototype = p;

return new f();

}

# Constructor and Method Chaining

If a method of B overrides a method of A, the overriding method in B may sometimes want to invoke the overridden method in A: this is called method chaining. Similarly, the subclass constructor B() may sometimes need to invoke the superclass constructor A(). This is called constructor chaining.

When we define a subclass, we only want to augment or modify the behavior of our superclass methods, not replace them completely. To do this, the constructor and methods of the subclass call or chain to the superclass constructor and the superclass methods.

/\*

\*

\* NonNullSet is a subclass of Set that does not allow null and undefined as members of the set.

\*/

function NonNullSet(){

//Just chain to our superclass.

//Invoke the superclass constructor as an ordinary function to

//initialize the object that has been created by this constructor

//invocation

Set.apply(this,arguments);

}

//Make NonNullSet a subclass of Set:

NonNullSet.prototype = inherit(Set.prototype);

NonNullSet.prototype.constructor = NonNullSet;

//To exclude null and undefined, we only have to override the add() method

NonNullSet.prototype.add = function(){

//check for null or undefined arguments

for(var i = 0; i < arguments.length; i++)

if (arguments[i] == null)

throw new Error("Can't add null or undefined to a NonNullSet");

return Set.prototype.add.apply(this, arguments);

}

# Preventing Class Extensions

It is usually considered a feature of JavaScript that classes can be dynamically extended by adding new methods to the prototype object. ECMAScript 5 allows you to prevent this, if you want to. Object.preventExtensions()makes an object nonextensible which means no new properties can be added to it. Object.seal()takes this a step further: it prevents the addition of new properties and also makes all current properties nonconfigurable, so they cannot be deleted. (A nonconfigurable property can still be writable, however, and can still be converted into a read-only property.) To prevent extensions to Object.prototype you can simply write:

Object.seal(Object.prototype);

Another dynamic feature of JavaScript is the ability to replace (or “monkey-patch”) methods of an object:

var original\_sort\_method = Array.prototype.sort;

Array.prototype.sort = function(){

var start = new Date();

original\_sort\_method.apply(this, arguments);

var end = new Date();

console.log(“Array sort took “ + (end – start) + “ milliseconds.” );

};

You can prevent this kind of alteration by making your instance methods read-only. Object.freeze() does everything Object.seal() and also makes the properties read-only and nonconfigurable.

There is a feature of read-only properties that is important to understand when working with classes. If an object o inherits a read-only property p, an attempt to assign to o.p will fail and will not create a new property in o. If you want to override an inherited read-only property, you have to use Object.defineProperty() or Object.defineProperties() or Object.create() to create the new property. This means that if you make the instance methods of a class read-only, it becomes significantly more difficult for the subclasses to override those methods.

# Execution of JavaScript Programs

There is no formal definition of a program in client-side JavaScript. We can say that a JavaScript program consists of all the JavaScript code in a web page (inline scripts, HTML Event Handlers, and javascript:URLs) along with external JavaScript code referenced with the src attribute of a <script> tag. All of these separate bits of code share a single global Window object. That means that they all see the same Document object, and they share the same set of global functions and variables: if a script defines a new global variable of function, that variable or function will be visible to any JavaScript code that runs after the script does.

If a web page includes an embedded frame (using the <iframe> element), the JavaScript code in the embedded document has a different global object than the code in the embedding document and it can be considered a separate JavaScript program. Remember, remember though, that there is no formal definition of what the boundaries of a JavaScript program are. If the container document and the contained document are from the same server, the code in one document can interact with the code in the other, and you can treat them as two interacting parts of a single program if you wish.

JavaScript program execution occurs in two phases. In the first phase, the document content is loaded and the code from <script> elements (both inline scripts and external scripts) is run. Scripts generally run in the order in which they appear in the document. The JavaScript code is run from top to bottom, in the order that it appears, subject, of course to JavaScript’s conditional loops and other control statements.

Once the document is loaded and all the scripts have run, JavaScript execution enters into the second phase. This phase is asynchronous and event driven. During this event-driven phase, the web browser invokes event handler functions in response to events that occur asynchronously.

Both core and client JavaScript have a single-threaded execution model. Scripts and event handlers are executed one at a time without concurrency. This keeps JavaScript programming simple. Single threaded execution means that web browsers must stop responding to user input while scripts and event handlers are executing. This places a burden on JavaScript programmers: it means that JavaScript scripts and event handlers must not run for too long or perform a computationally intensive task that will introduce a delay into document loading making the browser unresponsive.

# Security

## The Same Origin Policy

The same-origin policy is a sweeping security restriction on what web content JavaScript code can interact with. It typically comes into play when a web page includes <iframe> elements or opens other browser windows. In this case, the same-origin policy governs the interactions of JavaScript code in one window or frame with the content of other windows and frames. Specifically, a script can read only the properties of windows and documents that have the same origin as the document that contains the script.

The origin of a document is defined as the protocol, host and port of the URL, from which the document was loaded. Documents loaded from different web servers have different origins. Documents loaded through different ports of the same host have different origins. And a document loaded with the http: protocol has a different origin than one loaded with the https: protocol, even if they come from the same web server.

It is important to understand that the origin of the script itself is not relevant to the same-origin policy: What matters is the origin of the document in which the script is embedded. Suppose, for example, that a script hosted by host A is included (using the src property of a <script> element) in a web page served by host B. The origin of that script is host B and the script has full access to the content of the document that contains it. If the script opens a new window and loads a second document from host B, the script also has full access to the content of that second document. But if the script opens a third window and loads a document from host C (or even from host A) into it, the same-origin policy comes into effect and prevents the script from accessing this document.

The same-origin policy does not actually apply to all properties of all objects in a window from a different origin. But it does apply to many of them, and, in particular, it applies to practically all the properties of the Document object. You should consider any window or frame that contains a document from another server to be off limits to your scripts. If your script opened the window, your script can close it, but it cannot “look inside” the window in any way. The same-origin policy also applies to scripted HTTP requests made with XMLHttpRequest object. This object allows client-side JavaScript code to make arbitrary HTTP requests to the web server from which the containing document was loaded, but it does not allow scripts to communicate with other web servers.

Client Side JavaScript

# Document Elements as Window Properties

If you name an element in your HTML document using the id attribute and if the Window object does not already have a property by that name, the Window object is given an non-enumerable property whose name is the value of the id attribute and whose value is the HTMLElement object that represents the document element.

The implicit use of element IDs as global variables is a historical quirk of web browser evolution. It is required for backward compatibility with existing web pages, but its use is not recommended-anytime a browser vendor defines a new property of the Window object, it breaks any code that uses an implicit definition of that property name. Instead used document.getElementById() to look up elements explicitly.

Any HTML element with an id attribute will become the value of a global variable assuming the ID is not already used by the Window object. The following HTML elements also behave this way when given a name attribute:

<a> <applet> <area> <embed> <form> <frame> <frameset> <iframe> <img> <object>

The id element is required to be unique within a document but you can have more than one name attribute with the same value. If more than one of the elements above has the same name attributed or if one element has a name attribute and another element has an id with the same value, the implicit global variable with that name will refer to an array like object that holds each of the named elements.

There is a special case for <iframe> elements with a name or id attribute. The implicitly created variable for these elements refers not to the Element object that represents the element itself, but to the Window object that represents the nested browser frame created by the <iframe> element.

# Multiple Windows and Frames

A single web browser window on your desktop may contain several tabs. Each tab is an independent browsing context. Each has its own Window object, and each is isolated from all the others. The scripts running in one tab usually have no way of even knowing that the other tabs exist, much less of interacting with their window objects or manipulating their document content.

But windows are not always isolated from one another. A script in one window or tab can open new windows or tabs, and when a script does this, the windows can interact with one another and with one another’s documents (subject to the constraints of the same-origin policy)

HTML documents may contain nested documents using an <iframe> element. An <iframe>

Creates a nested browsing context represented by a Window object of its own. The deprecated <frameset> and <frame> elements also create nested browsing contexts, and each <frame> is represented by a Window. Client side JavaScript makes very little distinction between windows, tabs, iframes and frames: they are all browsing contexts, and to JavaScript they are all Windows objects. Nested browsing contexts are not isolated from one another the way independent tabs usually are. A script running in one frame can always see its ancestor and descendant frames, though the same origin policy may prevent the script from inspecting the documents in those frames.

Since the Window is the global object of client-side JavaScript, each window or frame has a separate JavaScript execution context. Nevertheless, JavaScript code in one window can, subject to same origin constraints, use the objects, properties, and methods defined in other windows. When the same origin policy prevents the scripts in two distinct windows from interacting directly, HTML5 provides an event based message passing API for indirect communication.

# Opening and Closing Windows

You can open a new web browser window or tab with the open() method of the Window object. Window.open() loads a specified URL into a new or existing window and returns the Window object that represents that window. It takes four optional arguments.

The first argument to open() is the URL of the document to display in the new window. If this argument is omitted, the special blank-page URL about:blank is used.

The second argument to open() is a string that specifies a window name. If a window by that name already exists (and if the script is allowed to navigate to that window), that existing window is used. Otherwise a new window is created and is assigned the specified name. If this argument is omitted, the special name “\_blank” is used: it opens a new unnamed window.

Note that scripts cannot simply guess window names and take over the windows in use by other web applications: they can only name existing windows that they are ”allowed to navigate”. Loosely, a script can specify an existing window by name only if that window contains a document from the same origin or if the script opened that window (or recursively opened a window that opened that window). Also, if one window is a frame nested within the other, a script in either one can navigate the other. In this case, the reserved names “\_top” and “\_parent” can be useful.

The third optional argument to open() is a comma separated list of size and features attributes for the new window to be opened. If you omit this argument, the new window is given a default size and has a full set of UI components. The third argument is non-standard and the HTML5 specification insists that browsers be able to ignore it.

The fourth argument to open() is useful only when the second argument names an existing window. This fourth argument is a Boolean value that indicates whether the URL specified as the first argument should replace the current entry in the window’s browsing history (true) or create a new entry in the window’s browsing history (false).

## Window Names

The name of a window is important because it allows the open() method to refer to existing windows, and also because it can be used as the value of the HTML target attribute on <a> and <form> elements to indicate that the linked document (or the result of submitting the form) should be displayed in the named window. The target attribute on these elements can also be set to \_blank, \_parent, or \_top to direct the linked document into a new blank window, the parent window or frame, or the top-level window.

The name property of a Window object holds its name, if it has one. This property is writable and scripts can set it as desired. If a name(other than \_blank) is passed to Window.open(), the window created by that call will have the specified name as the initial value of its name property. If an <iframe> element has a name attribute, the Window object that represents the frame will use that name attribute as the initial value of the name property.

# Relationships between Frames

The open method of a Window object returns a new Window object that has an opener property that refers back to the original window. In this way, the two windows can refer to each other, and each can read properties and invoke methods of the other. A similar thing is possible with frames. Code running in a window or frame can refer to the containing window or frame and to nested child frames.

JavaScript code in any window or frame can refer to its own Window object as window or self. A frame can refer to the window object of the window or frame that contains it using the parent property.

A Window object that represents a top-level window or tab has no container, and its parent property simply refers to the window itself.

If a frame is contained within another frame that is contained within a top-level window, that frame can refer to the top-level window as parent.parent. The top property is a general-case shortcut, however: no matter how deeply a frame is nested, its top property refers to the top-level containing window.

The parent and top properties allow a script to refer to its frame’s ancestors. There is more than one way to refer to the descendant frames of a window or frame. Frames are created with <iframe> elements. You can obtain an Element object that represents an <iframe> just as you would do for any other element. Suppose your document contains <iframe id=”f1”>. Then the Element object that represents this iframe is:

var iframeElement = document.getElementById(“f1”);

<iframe> elements have a contentWindow property that refers to the Window object of the frame, so the Window object for this frame is:

var childFrame = document.getElementById(“f1”).contentWindow;

You can go in the reverse direction from the Window that represents a frame to the <iframe> Element that contains the frame with the frameElement property of the Window. Window objects that represent top-level windows rather than frames have a null frameElement property:

var elt = document.getElementById(“f1”);

var win = elt.contentWindow;

win.frameElement === elt; //always true for frames

window.frameElement === null; //for top level windows

It is not necessary to use the getElementById() method and the contentWindow property to obtain references to the child frames of a window, however. Every Window object has a frames property that refers to the child frames contained within the window or frame. The frames property refers to an array like object that can be indexed numerically or by frame name. Note that the elements of the frames[] array are Window objects not <iframe> elements.

If you specify the name or id attribute of an <iframe> element, that frame can be indexed by name as well as by number. A frame named “f1” would be frames[“f1”] or frames.f1 for example.

The names or IDs of <iframe> and other elements are automatically used as properties of the Window object and that <iframe> elements are treated differently than other elements: for frames, the value of these automatically created properties refer to a Window object rather than an Element object. This means we can refer to a frame named “f1” as f1 instead of frames.f1. HTML5 specifies that the frames property is a self-referential property just like window and self, and that it is the Window object itself that acts like an array of frames. This means we can refer to the first child frame as window[0] and we can query the number of frames with window.length or just length. Note that not all browsers make frame==window.

You can use the name or id attribute of an <iframe> element to give the frame a name that can be used in JavaScript code. If you use the name attribute, however, the name you specify also becomes the value of the name property of the Window that represents the frame. A name specified in this way can be used as the target attribute of a link, and it can be used as the second argument to window.open().

# JavaScript in Interacting Windows

Each window or frame is its own JavaScript execution context with a Window as its global object. But if code in one window or frame can refer to another window or frame and if the same origin policy does not prevent it, the scripts in one window or frame can interact with the scripts in another.

Imagine a web page with two <iframe> elements named A and B, and suppose that those frames contain documents from the same server and that those documents contain interacting scripts. The script in frame A might define a variable i:

var i = 3;

The script in frame B can refer to this variable as:

parent.A.i;

Similarly frame A can invoke a function f defined in B as:

parent.B.f();

or

var f = parent.B.f;

f();

Remember though that JavaScript uses lexical scoping which means that a function is executed in the scope in which it was defined, not the scope in which it was invoked. Thus if the function f refers to global variables, those variables are looked up as properties of frame B even when the function is invoked from frame A.

Remember that constructors are also functions so when you define a class with a constructor function and an associated prototype object that class is defined only within a single window. Suppose that the window contains frames A and B.

Scripts within the top level window can create new Set objects as:

var s = new Set();

Scripts in either frames must explicitly refer to the Set() constructor as a property of the parent window:

var s = new parent.Set();

or

var Set = top.Set;

var s = new Set();

Unlike user-defined classes, the built in classes like String, Date and RegExp are automatically predefined in all windows. This means, however, that each window has an independent copy of the constructor and an independent copy of the prototype object. For example, each window has its own copy of the String() constructor and String.prototype object. So if you write a new method for manipulating JavaScript strings and then make it a method of the String class by assigning it to the String.prototype object in the current window, all Strings created by code in that window can use the new method, However, the new method is not accessible to strings created in other windows.

The fact that each Window has its own prototype objects means that the instanceof operator does not work across windows. instanceof will evaluate to false, for example when used to compare a string from frame B to the String() constructor from frame A.

# Selecting Document Elements

The DOM defines a number of ways to select elements; you can query a document for an element or elements:

* with a specified id attribute
* with a specified name attribute
* with the specified tag name
* with the specified CSS class or classes
* matching the specified CSS selector

## Selecting Elements by Name

Unlike the id attribute, the name attribute is only valid on a handful of HTML elements including forms, form elements, <iframe> and <img> elements. Setting the name element of the following:

<a> <applet> <area> <embed> <form> <frame> <frameset> <iframe> <img> <object>

Automatically creates properties with those names in the Window object. A Similar thing is true for the Document object. Setting the name attribute of

<form> <img> <applet> <iframe> <embed> <object>

creates a property of the Document object whose name is the value of the attribute assuming of course, that the document does not already have a property with that name.

If there is only a single element with a given name, the value of the automatically created document property is the element itself. If there is more than one element, then the value of the property is a NodeList object that acts as an array of elements.

## Selecting Elements by Type

You can select all HTML or XML elements of a specified type or tag using the getElementsByTagName() method of the document object. To obtain a read-only array like object containing all span tags in a document do:

var spans = document.getElementByTagName(“span”);

Like getElementsByName(), getElementsByTagName() returns a NodeList object. The elements of the returned NodeList are in document order.

You can obtain a NodeList that represents all elements in a document by passing the wildcard argument “\*” to getElementsByTagName() method.

The Element class also defines a getElementsByTagName() method. It works the same way as the Document version, but it only selects elements that are descendants of the element on which it is invoked. So to find all <span> elements inside the first <p> element of a document you could write:

var firstpara = document.getElementsByTagName(“p”)[0];

var firstParaSpans = firstpara.getElementsByTagName(“span”)

For historical reasons, the HTMLDocument class defines shortcut properties to access certain kinds of nodes. The images, forms, links, embeds and plugins, anchors and scripts properties refer to read only arrays of <img>, <form> <a>, <embed>, <a> (with name attributes instead of href) and <script> elements respectively. These properties refer to HTMLCollection objects which are much like NodeList objects, but they can additionally be indexed by element ID or name.

HTMLDocument also defines two properties that refer to special elements rather than element collections; document.body and document.head. These properties are always defined.

# NodeLists and HTMLCollections

getElementsByName() and getElementsByTagName() return NodeList objects and properties like document.images and document.forms are HTMLCollection objects.

These objects are read-only array like objects. They have length properties and can be indexed (for reading but not writing) like true arrays. You can iterate the contents of a NodeList or HTMLCollection with a standard loop like this:

for(var i=0; i<document.images.length; i++) //loop through the images

document.images[i].style.display = “none”; //and hide them

You cannot invoke Array methods on NodeLists and HTMLCollections directly, but you can do so indirectly:

var content = Array.prototype.map.call(document.getElementsByTagName(“p”),

function(e){return e.innerHTML; });

One of the most important and surprising features of NodeList and HTMLCollection is that they are not static snapshots of a historical document state but are generally live and the list of elements they contain can vary as the document changes.

Usually, the liveness of NodeLists and HTMLCollections is quite helpful. If you will be adding or removing elements from the document while iterating through a NodeList, however, you may want to make a static copy of the NodeList first:

var snapshot = Array.prototype.slice.call(nodelist, 0);

# Selecting Elements by CSS class

Like getElementsByTagName(), getElementsByClassName() can be invoked on both HTML documents and HTML elements and it returns a live NodeList containing all matching descendants of the document or element. getElementsByClassName() takes a single string argument, but the string may specify multiple space-separated identifiers. Only elements that include all of the specified identifiers in their class attribute are matched. The order of the identifiers does not matter. Note that both the class attribute and the getElementsByClassName() methods separate class identifiers with spaces, not with commas.

//Find all descendants of the element named “log” that have the class

//”error” and the class “fatal”

var log = document.getElementById(“log”);

var fatal = log.getElementsByClassname(“fatal error”);

# Selecting Elements with CSS Selectors

The document querySelectorAll() method, takes a single string argument containing a CSS selector and returns a NodeList that represents all elements in the document that match the selector. Unlike previously described element selection methods, the NodeList returned by querySelectorAll() is not live: it holds the elements that match the selector at the time the method was invoked but it does not update as the document changes. If no elements match, querySelectorAll() returns an empty NodeList. If the selector string is invalid, querySelectorAll() throws an exception.

In addition to querySelectorAll(), the document object also defines querySelector() which is like querySelectorAll(), but returns only the first (in document order) matching element or null if there is no matching element.

These two methods are also defined on Elements (and also on DocumentFragment nodes). When invoked on an element, the specified selector is matched against the entire document and then the result set is filtered so that it only includes descendants of the specified element.

querySelectorAll() is the ultimate element selection method. The jQuery library uses this kind of CSS selector based query as its central programming paradigm. Web applications based on jQuery use a portable, cross-browser equivalent to querySelectorAll() named $().

# Document Structure and Traversal

Once you have selected an Element from a Document, you sometimes need to find structurally related portions (parent, siblings, children) of the document.

## Documents as Trees of Nodes

The Document object, its Element objects and the Text objects that represent runs of text in the document are all Node objects. Node defines the following important properties:

* parentNode
* childNodes
* firstChild, lastChild
* nextSibling, previousSibling
* nodeType  
  The kind of node this is. Document nodes have the value 9. Element nodes have the value 1. Text nodes have the value 3. Comment nodes are 8 and DocumentFragment nodes are 11.
* nodeValue  
  The textual content of a Text or Comment node
* nodename  
  The tag name of an Element, converted to uppercase

e.g. <html><head><title>Test</title></head><body>Hello World!</body></html>

document.childNodes[0].childNodes[1]

document.firstChild.firstChild.nextSibling

Both statements above select the <body> element.

Note, however, that this API is extremely sensitive to variations in the document text. If the document is modified by inserting a single newline between the <html> and the <head> tag, for example, the Text node that represents the newline becomes the first child of the first child, and the second child is the <head> element instead of the <body> element.

## Documents as Trees of Elements

When we are primarily interested in the Elements of a document instead of the text within them (and the whitespace between them), it is helpful to use an API that allows us to treat a document as a tree of Element objects, ignoring Text and Comment nodes that are also part of the document.

The first part of this API is the children property of Element objects. This is an HTMLCollection object. Unlike childNodes, however, the children list contains only Element objects.

Note that Text and Comment nodes cannot have children which means that the Node.parentNode property described above never returns a Text or Comment node. The following is the API for an element based document traversal:

* firstElementchild, lastElementChild
* nextElementSibling, previousElementSibling
* childElementCount

# HTML Attributes as Element properties

The HTMLElement objects that represent the elements of an HTML document define read/write properties that mirror the HTML attributes of the elements. HTML attributes are not case sensitive, but JavaScript property names are. To convert an attribute name to the JavaScript property, write it in lowercase. If the attribute is more than one word long, however, put the first letter of each word after the first in uppercase. Some HTML attribute names are reserved words in JavaScript. For these, the general rule is to prefix the property name with “html”; the one exception to this rule is the HTML class attribute which becomes className in JavaScript code.

# Getting and Setting non HTML attributes

The element type also defines getAttribute() and setAttribute() methods that you can use to query and set nonstandard HTML attributes and to query and set attributes on the elements of an XML document. Attribute values are all treated as strings, getAttribute() never returns a number, Boolean, or object. Second, these methods use standard attribute names, even when those names are reserved words in JavaScript e.g. image.getAttribute(“class”);

Element also defines two related methods, has Attribute() and removeAttribute(), which check for the presence of a named attribute and remove an attribute entirely.

# Dataset Attributes

It is sometimes useful to attach additional information to HTML elements, typically when JavaScript code will be selecting those elements and manipulating them in some way. You can use the getAttribute() and setAttribute() methods to read and write the values of non-standard attributes. The price you pay, however, is that your document will not be valid HTML.

HTML5 provides a solution where any attribute whose name is lowercase and begins with the prefix “data-“ is considered valid.

HTML 5 also defines a dataset property on Element objects. This property refers to an object, which has properties that correspond to the data- attributes with their prefix removed. Thus dataset.jqueryTest would hold the value of the data-jquery-test attribute.

Note, that the dataset property is a live, two-way interface to the data- attributes of an element. Setting or deleting a property of dataset sets or removes the corresponding data- attribute of the element.

# Attributes as Attr Nodes

The Node type defines an attributes property. This property is null for any nodes that are not Element objects. For element objects, attributes is a read-only array-like object that represents all the attributes of the element. The attributes object is live in the way that NodeLists are.

The values obtained when you index the attributes object are Attr objects. Attr objects are a specialized kind of Node but are never really used like one. The name and value properties of an Attr return the name and value of the attribute.

# Element Content

Look at the following HTML Fragment and ask yourself what the content of <p> is.

<p>This is a <i>simple</i> document.</p>

There are three ways to answer this:

* The content is the HTML string “This is a <i>simple</i> document.”
* The content is the plain-text string “This is a simple document.”
* The content is a Text node, an Element node that has a Text node child and another Text node.

Each answer is valid. The following sections explain how to work with the HTML representation, the plain-text representation and the tree representation of the element content.

## Element Content as HTML

Reading the innerHTML property of an Element returns the content of that element as a string of markup. Setting this property on an element invokes the web browser’s parser and replaces the element’s current content with a parsed representation of the new string.

HTML5 says that innerHTML should work on Document nodes as well as Element nodes. HTML5 also standardizes a property named outerHTML. When you query outerHTML the string of HTML that is returned includes the opening and closing tags of the element on which you queried it. When you set outerHTML on an element, the new content replaces the element itself. outerHTML is defined only for Element nodes, not Documents.

Another feature standardized by HTML5 is the insertAdjacentHTML() method, which allows you to insert a string of arbitrary HTML markup adjacent to the specified element. The markup is passed as the second argument to this method and the precise meaning of adjacent depends on the value of the first argument. This first argument should be a string with one of the values “beforebegin”, “afterbegin”, “beforeend”, “afterend”.

e.g.

\*<div id=”target”>\*This is the element content\*</div>\*

beforebegin = first \*

afterbegin = second \*

beforeend = third \*

afterend = fourth \*

## Element Content as Plain Text

Sometimes you want to query the content of an element as plain text, or to insert plain text into a document (without having to escape the angle brackets and ampersands used in HTML markup). The standard way to do this is with the textContent property of Node:

var para = document.getElementsByTagName(“p”)[0]; //First <p> in document

var text = para.textContent; //Text is “This is a simple document.”

Para.textContent = “Hello World!”; //alter paragraph content

The textContent property is a straightforward concatenation of all Text node descendants of the specified element.

## Element Content as Text Nodes

Another way to work with the content of an element is as a list of child nodes, each of which may have its own set of children. When thinking about element content, it is usually the Text nodes that are of interest. The nodeValue property of a Node is read/write property and you can set it to change the content displayed by a Text node. Text is a subtype of CharacterData. CharacterData defines a data property which is the same text as nodeValue.

CharacterData also defines infrequently used methods for appending, deleting, inserting, and replacing text within a Text node.

# Creating, Inserting and Deleting Nodes

The Document type defines methods for creating Element and Text objects and the Node type defines methods for inserting, deleting and replacing nodes in the tree.

e.g.

//Asynchronously load and execute a script from a specified url

function loadasync(url){

var head = document.getElementsByTagName(“head”)[0]; //find document

//head

var s = document.createElement(“script”); //Create a script element

s.src = url; //Set its src attribute

head.appendChild(s); //Insert the <script> into head

}

## Creating Nodes

You can create new Element nodes with the createElement() method of the Document object. Pass the tag name of the element as the method argument.

Text nodes are created with a similar method called createTextNode:

var newnode = document.createTextNode(“text node content”);

Document defines other factory methods, such as the infrequently used createComment() as well. createDocumentFragment() is used to create document fragments.

Another way to create new document nodes is to make copies of existing ones. Every node has a cloneNode() method that returns a new copy of the node. Pass true to recursively copy all descendants as well, or false to only make a shallow copy.

## Inserting Nodes

Once you have a new node, you can insert it into the document with the Node methods appendChild() or insertBefore(). appendChild() is invoked on the Element node that you want to insert into, and it inserts the specified node so that it becomes the lastChild of that node.

insertBefore() is like appendChild(), but it takes two arguments. The first argument is the node to be inserted. The second argument is the node before which that node is to be inserted. The second argument is the node before which that node is to be inserted. This method is invoked on the node that will be the parent of the new node, and the second argument must be a child of that parent node. If you pass null as that second argument, the insertBefore() behaves like appendChild() and inserts at the end.

If you call appendChild() or insertBefore() to insert a node that is already in the document, that node will automatically be removed from its current position and reinserted at its new position: there is no need to explicitly remove the node.

## Removing and Replacing Nodes

The removeChild() method removes a node from the document tree. Be careful however: this method isn’t invoked on the node to be removed but on the parent of that node. Invoke the method on the parent node and pass the child node that is to be removed as the method argument.

replaceChild() removes one child node and replaces it with a new one. Invoke this method on the parent node, passing the new node as the first argument and the node to be replaced as the second argument.

## Using DocumentFragments

A DocumentFragment is a special kind of Node that serves as a temporary container for other nodes. Create a DocumentFragment like this:

var frag = document.createDocumentFragment();

Like a Document node, a DocumentFragment stands alone and is not part of any other document. Its parentNode is always null. Like an Element, however, a DocumentFragment can have any number of children, which you can manipulate with appendChild(), insertBefore() and so on.

The special thing about DocumentFragment is that it allows a set of nodes to be treated as a single node: if you pass a DocumentFragment to appendChild(), insertBefore() or replaceChild(), it is the children of the fragment that are inserted into the document, not the fragment itself. (The children are moved from the fragment into the document and the fragment becomes empty and ready for reuse.)

# HTML Forms

## Selecting Forms and Form Elements

Forms and elements they contain can be selected from a document using standard methods like getElementById() and getElementsByTagName():

var fields = document.getElementById(“address”).getElementsByTagName(“input”);

In browsers that support querySelectorAll(), you might select all radio buttons or all elements with the same name, from a form with code like this:

//All radio buttons in the form with id “shipping”

document.querySelectorAll(“#shipping input[type=”radio”]’);

//All radio buttons with name “method” in form with id “shipping”

document.querySelectorAll(‘#shipping input[type=”radio”][name=”method”]’);

A form element with a name=”address” can be selected in any of these ways:

window.address //brittle: don’t use

document.address //only works for forms with name attribute

document.forms.address //explicit access to a form with name or id

document.forms[n] //brittle: n is the form’s numerical position

*(For historical reasons, the HTMLDocument class defines shortcut properties to access certain kinds of nodes. The images, forms, links, embeds and plugins, anchors and scripts properties refer to read only arrays of <img>, <form> <a>, <embed>, <a> (with name attributes instead of href) and <script> elements respectively. These properties refer to HTMLCollection objects which are much like NodeList objects, but they can additionally be indexed by element ID or name. See page 30)*

document.forms is an HTMLCollection object that allows form elements to be selected by numerical order, by id, or by name. Form objects themselves act like HTMLCollections of form elements and can be indexed by name or number. If a form with name address has a first element with name “street”, you can refer to that form element with any of these expressions:

document.forms.address[0]

document.forms.address.street

document.address.street //only for name=”address”, not id=”address”

If you want to be explicit about selecting a form element, you can index the elements property of the form object instead:

document.forms.address.elements[0]

document.forms.address.elements.street

*(The elements property is an HTMLCollection of the form elements and sometimes it is necessary to refer to elements using this collection. Look at the example below)*

The id attribute is the generally preferred way to name specific document elements. The name attribute, however, has a special purpose for HTML form submission and is commonly used with forms than with other elements. Remember that when you index an HTMLCollection with a name and more than one element shares that name, the returned value is an array-like object that contains all matching elements.

e.g.

<form name="shipping">

<fieldset><legend>Shipping Method</legend>

<label><input type="radio" name="method" value="1st">First-class</label>

<label><input type="radio" name="method" value="2day">2-day Air</label>

<label><input type="radio" name="method" value="overnite">Overnight</label>

</fieldset>

</form>

With this form you might refer to the array of radio button elements like this:

var methods = document.forms.shipping.elements.method;

Note that <form> elements have an HTML attribute and a corresponding JavaScript property named “method” *(i.e. a form has an html attribute called method and a corresponding JavaScript property method. This attribute in <form> is used to specify the request type i.e. POST)*, so in this case, we must use the elements property of the form instead of directly accessing the method property.

## Form and Element Properties

The JavaScript Form object supports two methods, submit() and reset(). Invoking the submit() method of a Form submits the form and invoking reset() resets the form elements.

All form elements have the following properties in common.

* type  
  a read only string that identifies the type of the form element.
* form  
  a read only reference to the Form object in which the element is contained, or null if the element is not contained within a <form> element.
* name  
  a read only string specified by the HTML name attribute
* value  
  a read/write string that specifies the value contained or represented by the form element. This is the string that is sent to the web server when the form is submitted.

## Form and element Event Handlers

Each Form element has an onsubmit event handler to detect form submission and an onreset event handler to detect form resets. The onsubmit handler is invoked just before the form is submitted; it can cancel the submission by returning false. The onsubmit handler is triggered only by a genuine click on a Submit button. Calling the submit() method of a form does not trigger the onsubmit handler. The same is true of the onreset event handler.

Form elements that are buttons fire a click event when activated (even when this activation happens through the keyboard rather than via an actual mouse click).

Other form elements fire a change event when the user changes the value represented by the element. This happens when the user enters text in a text field or selects an option from a drop down list. Note that this event is not fired every time the user types a key in a text field. It is fired only when the user changes the value of an element and then moves the input focus to some other form element. That, is the invocation of this event handler indicated a completed change. Radio buttons and checkboxes are buttons that have a state, and they fire both click and change events; the change event is the more useful of the two.

Form elements also fire a focus event when they receive keyboard focus and a blur event when they lose it.

An important thing to know about event handlers is that within the code of an event handler, the this keyword refers to the document element that triggered the event. Since elements within a form element have a form property that refers to the containing form, the event handlers of these elements can always refer to the Form object as this.form.

# Overview of CSS

To define the visual appearance of HTML elements, specify the value of CSS properties. To do this, follow the name of a property with a colon and a value:

font-weight: bold

When multiple name value pairs are required use semicolons to separate them. Comments appear between /\*\*/. Using // doesn’t work.

margin-left: 10%; /\*left margin is 10% of page width\*/

text-indent: .5in;

## Ways of associating CSS styles with HTML Elements

### Inline style

e.g.

<p style=”margin: 20px; border: solid red 2px;”>

### Using StyleSheets

A style sheet associates a set of style properties with sets of HTML elements that are described using selectors. A selector specifies or selects one or more elements of a document based on element ID, class or tag name or on more specialized criteria. The basic element of a css stylesheet is a style rule, which consists of a selector followed by a set of CSS properties and their values, enclosed in curly braces.

P {

text-indent: .5in;

}

.warning{ /\*Any element with class=warning\*/

background-color: yellow;

border: solid black 5px;

}

A CSS stylesheet can be associated with an HTML document by enclosing it within <style> and </style> tags within the <head> of a document. Like <script> element, the <style> element parses its content specially and does not treat it like HTML:

When a stylesheet is to be used by more than one page on a website, it is usually better to store it in its own file without any enclosing HTML tags. The CSS file can then be included in the HTML page using a <link> tag in the <head> of a document.

<link rel=”stylesheet” href=”mystyles.css” type=”text/css”>

## The Cascade

This term indicates that the style rules that apply to any given elements in a document can come from a cascade of different sources:

* The Web browser’s default stylesheet
* The document’s stylesheet
* The style attribute of individual HTML elements

Styles from the style attribute override styles from stylesheets and styles from a document’s stylesheets override the browser’s default styles.

The visual presentation of any given element may be a combination of style properties from all three sources. An element may even match more than one selector within a stylesheet, in which case the style properties associated with all of of those selectors are applied to the element. (If different selectors define different values for the same style property, the value associated with the most specific selector overrides the value associated with less specific selectors.

To display any document element, the web browser must combine the style attribute of that element with styles from all the matched selectors in the document stylesheets.

## Shortcut properties

Certain style properties that are commonly used together can be combined using special shortcut properties. For example border-left, border-right, border-top and border-bottom properties specify the border of each side and can be specified using a single border property with a compound value. Infact each of these properties is itself a shortcut. Instead of specifying border-top, you can specify border-top-color, border-top-style, and border-top-width.

## Non-standard properties

When browser vendors implement nonstandard CSS properties, they prefix the property names with a vendor specific string. Firefox uses -moz-, Chrome uses –webkit- and IE uses –ms-.

When working with CSS properties that have different names in different browsers, you may find it helpful to define a class for that property:

.radius10 {

border-radius: 10px;

-moz-border-radius: 10px;

-webkit-border-radius: 10px;

}

## Positioning Elements with CSS

The CSS positon property specifies the type of positioning applied to an element using four possible values for this property.

### static

This is the default value and specifies that the element is positioned according to the normal flow of document content which is usually top to bottom, left to right. Statically positioned elements cannot be positioned with top, left, and other properties. To use CSS positioning techniques with a document element, you must first set its position property to either absolute, fixed, or relative values.

### absolute

This value allows you to specify the position of an element relative to its containing element. Absolutely positioned elements are positioned independently of all other elements and are not part of the flow of statically positioned elements. An absolutely positioned element is positioned either relative to its nearest positioned ancestor or relative to the document itself.

### fixed

This valued allows you to specify an element’s position with respect to the browser window. Elements with fixed positioning are always visible and do not scroll with the rest of the document. Like absolutely positioned elements, fixed position elements are independent of all others and are not part of the document flow.

### relative

When the position property is set to relative, an element is laid out according to the normal flow, and its position is then adjusted to its position in the normal flow. The space allocated for the element in the normal document flow remains allocated for it, and the elements on either side of it do not close up to fill in that space, nor are they “pushed away” from the new position of the element.

Once you have set the position property of an element to something other than static, you can specify the position of that element with some combination of the left, top, right, and bottom properties.

If an element uses absolute positioning, its top and left properties are interpreted relative to the closest ancestor element that has its position property set to something other than static. If an absolutely positioned element has no positioned ancestor, the top and left properties are measured in document co-ordinates-they are offsets from the top-left corner of the document. If you wish to absolutely position an element relative to a container that is part of the normal document flow, use position:relative for the container and specify a top and left position of 0px. This makes the container dynamically positioned but leaves it at its normal place in the document flow. Any absolutely positioned children are then positioned relative to the container position.

In addition to the position of elements, CSS allows you to specify their size. This is most commonly done by providing values for width and height style properties.

<div style=”position: absolute; top: 10px; left: 10px; width: 10px; height: 10px, background-color: blue”>

</div>

Another way to specify the width of an element is to specify a value for both the left and right properties. Similarly, you can specify the height of an element by specifying both top and bottom. If you specify a value for left, right, and width, however, the width property overrides the right property; if the height of an element is overconstrained, height takes priority over bottom.

CSS requires position and dimension properties to be specified with a unit. You can use px, pixels, in, inches, cm, centimeters, pt, points, and em, ems which is a measure of the line height for the current font.

Instead of specifying absolute positions and sizes using the units shown above, CSS also allows you to specify the position and size of an element as a percentage of the size of the containing element. For example the following HTML creates an empty elment with a black border that is half as wide and half as high as the containing element (or browser window *I think this should be document and not window*) and centered within that element.

<div style=”position: absolute; left: 25%; top: 25%; width: 50%; height: 50%; border: 2px solid black”>

</div>

## The third dimension: z-index

The z-index allows you to specify the stacking order of elements and indictae which of two or more overlapping elements is drawn on top of the others. When two or more elements overlap, they are drawn in order from lowest to highest z-index; the element with the highest z-index appears on top of all the others. If overlapping elements have the same z-index, they are drawn in the order in which they appear in the document so that the last overlapping element appears on top.

Note that z-index stacking applies only to sibling elements (i.e. elements that are children of the same container). If two elements that are not siblings overlap, setting their individual z-index properties does not allow you to specify which one is on top. Instead you must specify the z-index property for the two sibling containers of the two overlapping elements.

Non positioned elements (i.e. elements with default position:static) are always laid out in a way that prevents overlaps, so the z-index property does not apply to them.

## Borders, Margins and Padding

CSS allows you to specify borders, margins and padding around any element. The border of an element is a rectangle drawn around it. CSS properties allow you to specify the style, color, and thickness of the border:

border: solid black 1px;

border: 3px dotted red;

It is possible to specify the border width, style, and color using individual CSS properties and it is also possible to specify the border for individual sides of an element. To draw a line beneath an element, for example, simply specify its border-bottom property. It is even possible to specify the width, style, or color of a single side of an element with properties such as border-top-width and border-left-color.

The margin and padding properties both specify blank space around an element. The important difference is that margin specifies space outside the border, between the border and adjacent elements, and padding specifies space inside the border, between the border and the element content. A margin provides visual space between a bordered element and its neighbors in the normal document flow. Padding keeps element content visually separated from its border. If an element has no border, padding is typically not necessary. If an element is dynamically positioned, it is not part of the normal document flow, and its margins are irrelevant.

You can specify the margin and paddings of an element with the margin and padding properties:

margin: 5px; padding 5px;

or you can specify margin and padding values for all four edges of an element with the margin and padding properties. You specify the top values first and then proceed clockwise: top, right, bottom, left. The following are equivalent ways of specifying the padding for each four sides of an element.

padding: 1px 2px 3px 4px;

Or

padding-top: 1px;

padding-right: 2px;

padding-bottom: 3px;

padding-left: 4px;

The margin property works in the same way.

## The CSS Box Model and Positioning Details

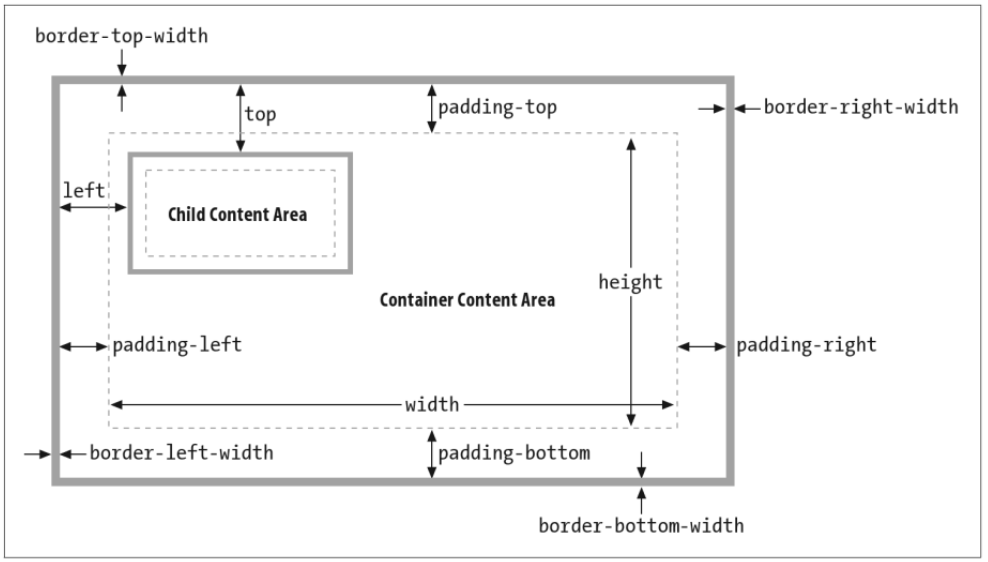


Figure The CSS box model: borders, padding, and positioning properties

Figure  1 shows an absolutely positioned element nested inside a positioned container element. Both the container and the contained elements have borders and padding, and the figure illustrates the CSS properties that specify padding and border width for each side of the container element. Notice that no margin properties are shown: margins aren’t relevant to absolutely positioned elements.

Figure 1 contains other, more important information as well. First, width and height specify the size of an element’s content area only; they do not include any additional space required for the element’s padding or border (or margins). To determine the full on-screen size of an element with a border, you must add the left and right padding and left and right border widths to the element width, and you must add the top and bottom padding and top and bottom border widths to the element’s height.

Second, the left and top properties specify the distance from the inside of the container’s border to the outside of the positioned element’s border. These properties do not measure from the upper-left corner of the content area of the container, but from the upper-left corner of the container’s padding. Similarly, the right and bottom properties measure from the lower-right corner of the padding.

Here’s an example that might make this clearer. Suppose you’ve created a dynamically positioned container element that has 10 pixels of padding all the way around its content area and a 5-pixel border all the way around the padding. Now suppose you dynamically position a child element inside this container. If you set the left property of the child to “0 px”, you’ll discover that the child is positioned with its left edge right up against the inner edge of the container’s border. With this setting, the child overlaps the container’s padding, which presumably was supposed to remain empty (since that is the purpose of padding). If you want to position the child element in the upper left corner of the container’s content area, you should set both the left and top properties to “10px”.

## The border-box model and the box-sizing property

The standard CSS box model specifies that the width and height style properties give the size of the content area and do not include padding and borders. We might call this box model the “content-box model.” There are exceptions to the content-box model in old versions of IE and also in new versions of CSS. Before IE6, and when IE6, 7, or 8 displays a page in “quirks mode” (when the page has no <! DOCTYPE > or has an insufficiently strict doctype), the width and height properties do include the padding and border widths.

IE’s behavior is a bug, but IE’s nonstandard box model is often quite useful. Recognizing this, CSS3 introduces a box-sizing property. The default value is content-box, which specifies the standard box model described above. If you instead specify box-sizing:border-box, the browser will use the IE box model for that element, and the width and height properties will include border and padding. The border-box model is particularly useful when you want to specify the overall size of an element as a percentage but also want to specify the border and padding size in pixels:

< div style =" box-sizing:border-box; width: 50%; padding: 10px; border: solid black 2px;" >

The box-sizing property is supported by all current browsers but is not yet universally implemented without a prefix. In Chrome and Safari, use -webkit-box-sizing. In Firefox, use -moz-box-sizing. In Opera and IE8 and later, you can use box-sizing without any prefix.

## Element Display and Visibility

Two CSS properties affect the visibility of a document element: visibility and display. The visibility property is simple: when the property is set to the value hidden, the element is not shown; when it is set to the value visible, the element is shown. The display property is more general and is used to specify the type of display an item receives. It specifies whether an element is a block element, an inline element, a list item, or so on. When display is set to none, however, the affected element is not displayed, or even laid out, at all.

The difference between the visibility and display style properties has to do with their effect on elements that use static or relative positioning. For an element that appears in the normal layout flow, setting visibility to hidden makes the element invisible but reserves space for it in the document layout. Such an element can be repeatedly hidden and shown without changing the document layout. If an element’s display property is set to none, however, no space is allocated for it in the document layout; elements on either side of it close up as if it were not there. The display property is useful, for example, when creating expanding and collapsing outlines.

visibility and display have equivalent effects when used with absolute- or fixed-position elements because these elements are not part of the document layout. The visibility property is generally preferred for hiding and showing positioned elements, however.

## Querying Computed Styles

The computed style for an element is the set of property values that the browser derives (or computes) from the inline style plus all applicable style rules in all linked stylesheets: it is the set of properties actually used to display the element. Like inline styles, computed styles are represented with a CSSStyleDeclaration object. Unlike inline styles, however, computed styles are read-only. You can’t set these styles, but the computed CSSStyleDeclaration object for an element lets you determine exactly what style property values the browser used when rendering that element.

Obtain the computed style for an element with the getComputedStyle() method of the Window object. The first argument to this method is the element whose computed style is desired. The second argument is required and is usually null or the empty string, but it can also be a string that names a CSS pseudoelement, such as “: before”, “: after”, “: first-line”, or “: first-letter”:

The return value of getComputedStyle() is a CSSStyleDeclaration object that represents all the styles that apply to the specified element (or pseudoelement). There are a number of important differences between a CSSStyleDeclaration object that represents inline styles and one that represents computed styles:

* Computed style properties are read-only.
* Computed style properties are absolute: relative units like percentages and points are converted to absolute values. Any property that specifies a size (such as a margin size or a font size) will have a value measured in pixels. This value will be a string with a “px” suffix, so you’ll still need to parse it, but you won’t have to worry about parsing or converting units.
* Properties whose values are colors will be returned in “rgb(#,#,#)” or “rgba(#,#,#,#)” format.
* Shortcut properties are not computed, only the fundamental properties that they are based on. Don’t query the margin property, for example, but use marginLeft, marginTop, and so on. The cssText property of the computed style is undefined.

# Handling Events

The event type is a string that specifies what kind of event occurred.

The event target is the object on which the event occurred or with which the event is associated.

An event handler or event listener is a function that handles or responds to an event. Applications register their event handler functions with the web browser, specifying an event type and an event target.

An event object is an object that is associated with a particular event and contains details about that event. Event objects are passed as an argument to the event handler function (except in IE8 and before where they are sometimes only available through the global variable event). All event objects have a type property that specifies the event type and a target property that specifies the event target. (In IE8 and before, use srcElement instead of target.)

Event propagation is the process by which the browser decides which objects to trigger event handlers on. For events that are specific to a single object (such as the load event on the Window object), no propagation is required. When certain kinds of events occur on document elements, however, they propagate or “bubble” up the document tree. If the user moves the mouse over a hyperlink, the mousemove event is first fired on the < a > element that defines that link. Then it is fired on the containing elements: perhaps a < p > element, a < div > element, and the Document object itself. It is sometimes more convenient to register a single event handler on a Document or other container element than to register handlers on each individual element you’re interested in. An event handler can stop the propagation of an event, so that it will not continue to bubble and will not trigger handlers on containing elements. Handlers do this by invoking a method or setting a property of the event object. In another form of event propagation, known as event capturing, handlers specially registered on container elements have the opportunity to intercept (or “capture”) events before they are delivered to their actual target.

Some events have default actions associated with them. When a click event occurs on a hyperlink, for example, the default action is for the browser to follow the link and load a new page. Event handlers can prevent this default action by returning an appropriate value, invoking a method of the event object, or by setting a property of the event object. This is sometimes called “canceling” the event.

## Legacy Event Types

The events you’ll use most often in your web apps are generally the ones that have been around the longest and are universally supported: events for dealing with the mouse, the keyboard, HTML forms, and the Window object. The sections below explain many important details about these kinds of events.

### Form Events

< form > elements fire submit events when the form is submitted and reset events when the form is reset. Button-like form elements (including radio buttons and checkboxes) fire click events when the user interacts with them. Form elements that maintain some kind of state generally fire change events when the user changes their state by entering text, selecting an item, or checking a box. For text input fields, a change event is not fired until the user has finished interacting with a form element and has tabbed or clicked to move focus to another element. Form elements respond to keyboard focus changes by firing focus and blur events when they gain and lose the focus.

The submit and reset events have default actions that can be canceled by event handlers, and some click events do, too. The focus and blur events do not bubble, but all the other form events do.

Finally, note that browsers other than IE trigger an input event on < textarea > and other text-input form elements whenever the user enters text (via the keyboard or cut-and-paste) into the element. Unlike the change event, these input events are triggered for each insertion. Unfortunately, the event object of an input event does not specify what text has been input.

### Window Events

The load event is the most important of these events: it is fired when a document and all of its external resources (such as images) are fully loaded and displayed to the user

DOMContentLoaded and readystatechange are alternatives to the load event: they are triggered sooner, when the document and its elements are ready to manipulate, but before external resources are fully loaded.

The unload event is the opposite of load: it is triggered when the user is navigating away from a document. An unload event handler might be used to save the user’s state, but it cannot be used to cancel navigation. The beforeunload event is similar to unload but gives you the opportunity to ask the user to confirm that they really want to navigate away from your web page. If a handler for beforeunload returns a string, that string will be displayed to the user in a confirmation dialog before the new page is loaded, and the user will have the opportunity to cancel her navigation and remain at your page.

Individual document elements, such as < img > elements, can also register handlers for load and error events. These are triggered when an external resource (the image, for example) is fully loaded, or when an error occurs that prevents it from loading.

The focus and blur events described above for form elements are also used as Window events: they are triggered on a window when that browser window receives or loses keyboard focus from the operating system.

Finally, the resize and scroll events are fired on a Window when the user resizes or scrolls the browser window. Scroll events can also be fired on any scrollable document element, such as those with the CSS overflow property (Partial Visibility: overflow and clip) set. The event object passed to resize and scroll event handlers is just an ordinary Event object and does not have properties that specify how much resizing or scrolling occurred — you can determine the new window size and scrollbar position using the techniques shown in Document and Element Geometry and Scrolling.

### Mouse Events

Mouse events are generated when the user moves or clicks the mouse over a document. These events are triggered on the most deeply nested element that the mouse pointer is over, but they bubble up through the document. The event object passed to mouse event handlers has properties set that describe the position and button state of the mouse and also specify whether any modifier keys were held down when the event occurred. The clientX and clientY properties specify the position of the mouse in window coordinates. The button and which properties specify which mouse button (if any) was pressed.

The altKey, ctrlKey, metaKey, and shiftKey properties are set to true when the corresponding keyboard modifier keys are held down. And for click events, the detail property specifies whether this was a single, double, or triple click.

The mousemove event is triggered any time the user moves or drags the mouse. These events occur frequently, so mousemove handlers must not trigger computationally intensive tasks. The mousedown and mouseup events are triggered when the user presses and releases a mouse button.

After a mousedown and mouseup event sequence, the browser also triggers a click event. The click event was described above as a device-independent form event, but it is actually triggered on any document element, not just form elements, and it is passed an event object with all of the extra mouse-related fields described above. If the user clicks a mouse button twice in a row (within a sufficiently short amount of time), the second click event will be followed by a dblclick event. Browsers often display a context menu when the right mouse button is clicked. They generally fire a contextmenu event before displaying the menu, and if you cancel the event, you can prevent the display of the menu. This is also an easy way to be notified of right mouse button clicks.

When the user moves the mouse so that it goes over a new element, the browser fires a mouseover event on that element. When the mouse moves so that it is no longer over an element, the browser fires a mouseout event on that element. For these events, the event object will have a relatedTarget property that specifies the other element involved in the transition.

mouseover and mouseout events bubble like all of the mouse events described here. This is often inconvenient, because when a mouseout handler is triggered, you have to check whether the mouse actually left the element you are interested in or if it merely transitioned from one child of the element to another.

When the user rotates the mouse wheel, browsers trigger a mousewheel event. The event object passed with these events includes properties that specify how much, and in which direction, the wheel was rotated.

### Key Events

When the web browser has keyboard focus, it generates events each time the user presses or releases a key on the keyboard. Keyboard shortcuts that have meaning to the operating system or to the browser itself are often “eaten” by the OS or browser and may not be visible to JavaScript event handlers, however. Keyboard events are triggered on whatever document element has keyboard focus, and they bubble up to the document and window. If no element has the focus, the events are triggered directly on the document. Keyboard event handlers are passed an event object with a keyCode field that specifies what key was pressed or released. In addition to keyCode, the event object for key events also has altKey, ctrlKey, metaKey, and shiftKey that describe the state of the keyboard modifier keys.

The keydown and keyup events are low-level keyboard events: they are triggered whenever a key (even a modifier key) is pressed or released. When a keydown event generates a printable character, an additional keypress event is triggered after the keydown but before the keyup. (In the case of a key that is held down until it repeats, there may be many keypress events before the keyup event.) The keypress event is a higher-level text event, and its event object specifies the character that was generated, not the key that was pressed.

### HTML5 Events

The HTML5 drag-and-drop API allows JavaScript applications to participate in OS-based drag-and-drop operations, transferring data between web applications and native applications.

HTML5 defines a history management mechanism (History Management) that allows web applications to interact with the browser’s Back and Forward buttons. This mechanism involves events named hashchange and popstate. These events are lifecycle notification events like load and unload and are fired at the Window object rather than any individual document element.

HTML5 defines a lot of new features for HTML forms. In addition to standardizing the form input event described earlier, HTML5 also defines a form validation mechanism, which includes an invalid event fired on form elements that have failed validation.

HTML5 includes support for offline web applications that can be installed locally in an application cache so that they can run even when the browser is offline (as when a mobile device is out of network range). The two most important events associated with this are the offline and online events: they are triggered on the Window object whenever the browser loses or gains a network connection.

A number of new web application APIs use a message event for asynchronous communication. The Cross-Document Messaging API (Cross-Origin Messaging) allows scripts in a document from one server to exchange messages with scripts in a document from another server. This works around the limitations of the same-origin policy (The Same-Origin Policy) in a secure way. Each message that is sent triggers a message event on the Window of the receiving document. The event object passed to the handler includes a data property that holds the content of the message as well as source and origin policies that identify the sender of the message. The message event is used in similar ways for communication with Web Workers (Web Workers) and for network communication via Server-Sent Events (Comet with Server-Sent Events) and WebSockets (Web Sockets).

HTML5 and related standards define some events that are triggered on objects other than windows, documents, and document elements. Version 2 of the XMLHttpRequest specification, as well as the File API specification, define a series of events that track the progress of asynchronous I/ O. They trigger events on an XMLHttpRequest or FileReader object. Each read operation begins with a loadstart event, followed by progress events and a loadend event. Additionally, each operation ends with a load, error, or abort event just before the final loadend event.

Finally, HTML5 and related standards define a few miscellaneous event types. The Web Storage (localStorage and sessionStorage) API defines a storage event (on the Window object) that provides notification of changes to stored data. HTML5 also standardizes the beforeprint and afterprint events that were originally introduced by Microsoft in IE. As their names imply, these events are triggered on a Window immediately before and immediately after its document is printed and provide an opportunity to add or remove content such as the date and time that the document was printed. (These events should not be used to change the presentation of a document for printing because CSS media types already exist for that purpose.)

## Registering Event Handlers

There are two basic ways to register event handlers. The first, from the early days of the Web, is to set a property on the object or document element that is the event target. The second, newer and more general, technique is to pass the handler to a method of the object or element. To complicate matters, there are two versions of each technique. You can set an event handler property in JavaScript code, or for document elements, you can set the corresponding attribute directly in HTML.

### Setting Event Handler Properties

The simplest way to register an event handler is by setting a property of the event target to the desired event handler function. By convention, event handler properties have names that consist of the word “on” followed by the event name: onclick, onchange, onload, onmouseover, and so on. Note that these property names are case sensitive and are written in all lowercase, even when the event type (such as “readystatechange” consists of multiple words.

// Set the onload property of the Window object to a function.

// The function is the event handler: it is invoked when the document loads. window.onload = function() {

// Look up a < form > element

var elt = document.getElementById(" shipping\_address");

// Register an event handler function that will be invoked right

// before the form is submitted.

elt.onsubmit = function() { return validate( this); }

}

The shortcoming of event handler properties is that they are designed around the assumption that event targets will have at most one handler for each type of event. If you are writing library code for use in arbitrary documents, it is better to register event handlers using a technique (such as addEventListener()) that will not modify or overwrite any previously registered handlers.

### Setting Event Handler Attributes

The event handler properties of a document element can also be set as attributes on the corresponding HTML tag. If you do this, the attribute value should be a string of JavaScript code. That code should be the body of the event handler function, not a complete function declaration. That is, your HTML event handler code should not be surrounded by curly braces and prefixed with the function keyword.

e.g.

<button onclick=”alert(‘thank you’);”>click here</button>

If an HTML event handler attribute contains multiple JavaScript statements, you must remember to separate those statements with semicolons or to break the attribute value across multiple lines.

Some event types are directed at the browser as a whole, rather than at any particular document element. In JavaScript, handlers for these events are registered on the Window object. In HTML, we place them on the < body > tag, but the browser registers them on the Window.

When you specify a string of JavaScript code as the value of an HTML event handler attribute, the browser converts your string into a function that looks something like this:

function( event){

with(document) {

with( this.form | | {}) {

with( this) {

/\* your code here \*/

}

}

}

}

## addEventListener()

In the standard event model supported by all browsers other than IE8 and earlier, any object that can be an event target — this includes the Window and Document objects and all document Elements — defines a method named addEventListener() that you can use to register an event handler for that target. addEventListener() takes three arguments.

* The first is the event type for which the handler is being registered. The event type (or name) is a string and it should not include the “on” prefix that is used when setting event handler properties.
* The second argument to addEventListener() is the function that should be invoked when the specified type of event occurs.
* The final argument to addEventListener() is a boolean value.

Normally, you’ll pass false for this argument. If you pass true instead, your function is registered as a capturing event handler and is invoked at a different phase of event dispatch.

You can call addEventListener() multiple times to register more than one handler function for the same event type on the same object. When an event occurs on an object, all of the handlers registered for that type of event are invoked, in the order in which they were registered.

Invoking addEventListener() more than once on the same object with the same arguments has no effect — the handler function remains registered only once, and the repeated invocation does not alter the order in which handlers are invoked.

addEventListener() is paired with a removeEventListener() method that expects the same three arguments but removes an event handler function from an object rather than adding it.

## attachEvent()

Internet Explorer, prior to IE9, does not support addEventListener() and removeEventListener(). In IE5 and later, it defines similar methods attachEvent() and detachEvent(). The attachEvent() and detachEvent() methods work like addEventListener() and removeEventListener(), with the following exceptions:

* Since the IE event model does not support event capturing, attachEvent() and detachEvent() expect only two arguments: the event type and the handler function.
* The first argument to the IE methods is an event handler property name, with the “on” prefix, rather than the unprefixed event type. For example, pass “onclick” to attachEvent() where you would pass “click” to addEventListener().
* attachEvent() allows the same event handler function to be registered more than once. When an event of the specified type occurs, the registered function will be invoked as many times as it was registered.

## Event Handler Invocation

### Event Handler Argument

Event handlers are normally (there is one exception, described below) invoked with an event object as their single argument. The properties of the event object provide details about the event. The type property, for example, specifies the type of the event that occurred.

In IE8 and before, event handlers registered by setting a property are not passed an event object when they are invoked. Instead, the event object is available through the global variable window.event.

Event handlers registered with attachEvent() are passed an event object, but they can also use window.event.

When you register an event handler by setting an HTML attribute, the browser converts your string of JavaScript code into a function. Browsers other than IE construct a function with a single argument named event. IE constructs a function that expects no argument. If you use the identifier event in such a function, you are referring to window.event. In either case, HTML event handlers can refer to the event object as event.

### Event Handler Context

Event handlers are invoked (with one IE-related exception, described below) as methods of the object on which they are defined. That is, within the body of an event handler, the this keyword refers to the event target.

Handlers are invoked with the target as their this value even when registered using addEventListener(). Unfortunately, however, this is not true for attachEvent(): handlers registered with attachEvent() are invoked as functions, and their this value is the global (Window) object.

### Event Handler Scope

Like all JavaScript functions, event handlers are lexically scoped. They are executed in the scope in which they are defined, not the scope from which they are invoked, and they can access any local variables from that scope.

Event handlers registered as HTML attributes are a special case, however. They are converted into top-level functions that have access to global variables but not to any local variables. But, for historical reasons, they run with a modified scope chain. Event handlers defined by HTML attributes can use the properties of the target object, the containing < form > object (if there is one), and the Document object as if they are local variables.

On the other hand, the modified scope chain of HTML event handlers is a source of pitfalls, since the properties of each of the objects in the chain shadow any properties of the same name in the global object. The Document object defines a (rarely used) open() method, for example, so an HTML event handler that wants to invoke the open() method of the Window object must explicitly write window.open instead of open. There is a similar (but more pernicious) problem with forms, because the names and IDs of form elements define properties on the containing form element (see Selecting Forms and Form Elements). So if a form contains an element with the ID “location”, for example, all HTML event handlers within that form must use window.location instead of location if they want to refer to the window’s Location object.

### Handler Return Value

The return value of an event handler registered by setting an object property or an HTML attribute is sometimes significant. In general, a return value of false tells the browser that it should not perform the default action associated with the event. The onclick handler of a Submit button in a form, for example, can return false to prevent the browser from submitting the form. (This is useful if the user’s input fails client-side validation.) Similarly, an onkeypress handler on an input field can filter keyboard input by returning false if the user types an inappropriate character.

The return value of the onbeforeunload handler of the Window object is also significant. This event is triggered when the browser is about to navigate to a new page. If this event handler returns a string, it will be displayed in a modal dialog box that asks the user to confirm that she wants to leave the page.

It is important to understand that event handler return values are significant only for handlers registered as properties. We’ll see below that event handlers registered with addEventListener() or attachEvent() must instead call the preventDefault() method or set the returnValue property of the event object.

### Invocation Order

A document element or other object may have more than one event handler registered for a particular type of event. When an appropriate event occurs, the browser must invoke all of the handlers, following these rules of invocation order:

* Handlers registered by setting an object property or HTML attribute, if any, are always invoked first.
* Handlers registered with addEventListener() are invoked in the order in which they were registered.
* Handlers registered with attachEvent() may be invoked in any order and your code should not depend on sequential invocation.

### Event Propagation

After the event handlers registered on the target element are invoked, most events “bubble” up the DOM tree. The event handlers of the target’s parent are invoked. Then the handlers registered on the target’s grandparent are invoked. This continues up to the Document object, and then beyond to the Window object. Event bubbling provides an alternative to registering handlers on lots of individual document elements: instead you can register a single handler on a common ancestor element and handle events there. You might register a “change” handler on a < form > element, for example, instead of registering a “change” handler for every element in the form.

Most events that occur on document elements bubble. Notable exceptions are the focus, blur, and scroll events. The load event on document elements bubbles, but it stops bubbling at the Document object and does not propagate on to the Window object. The load event of the Window object is triggered only when the entire document has loaded.

Event bubbling is the third “phase” of event propagation. The invocation of the event handlers of the target object itself is the second phase. The first phase, which occurs even before the target handlers are invoked, is called the “capturing” phase. Recall that addEventListener() takes a boolean value as its third argument. If that argument is true, the event handler is registered as a capturing event handler for invocation during this first phase of event propagation. Event bubbling is universally supported: it works in all browsers including IE, and it works for all handlers, regardless of how they are registered (unless they are registered as capturing event handlers). Event capturing, by contrast, only works with event handlers registered with addEventListener() when the third argument is true. This means that event capturing is not available in IE prior to IE9, and is not, at the time of this writing, a commonly used technique.

The capturing phase of event propagation is like the bubbling phase in reverse. The capturing handlers of the Window object are invoked first, then the capturing handlers of the Document object, then of the body object, and so on down the DOM tree until the capturing event handlers of the parent of the event target are invoked. Capturing event handlers registered on the event target itself are not invoked.

### Event Cancellation

Handler Return Value explained that the return value of event handlers registered as properties can be used to cancel the browser’s default action for the event. In browsers that support addEventListener(), you can also cancel the default action for an event by invoking the preventDefault() method of the event object. In IE prior to IE9, however, you do the same by setting the returnValue property of the event object to false.

Canceling the default action associated with an event is only one kind of event cancellation. We can also cancel the propagation of events. In browsers that support addEventListener(), the event object has a stopPropagation() method that you can invoke to prevent the continued propagation of the event. If there are other handlers defined on the same object, the rest of those handlers will still be invoked, but no event handlers on any other object will be invoked after stopPropagation() is called. The stopPropagation() method can be called at any time during event propagation. It works during the capturing phase, at the event target itself, and during the bubbling phase.

Prior to IE9, IE does not support the stopPropagation() method. Instead, the IE event object has a property named cancelBubble. Set this property to true to prevent any further propagation.

## Document Load Events

Most web applications need notification from the web browser to tell them when the document has been loaded and is ready to be manipulated. The load event on the Window object serves this purpose.

The load event does not fire until a document and all of its images are fully loaded. It is usually safe, however, to start running your scripts after the document is fully parsed but before images are downloaded. You can improve the startup time of your web applications if you trigger your scripts on events other than “load”.

The DOMContentLoaded event is fired when the document has been loaded and parsed and any deferred scripts have been executed.

As described in Client-Side JavaScript Timeline, the document.readyState property changes as the document loads. In IE, each change in state is accompanied by a readystatechange event on the Document object, and it is possible to use this event to determine when IE reaches the “complete” state. HTML5 standardizes the readystatechange event, but fires it immediately before the load event, so it is not clear that much advantage is gained by listening for “readystatechange” instead of “load”.