# **Course Reader**

# Karthik Viswanathan and Vivek Nair

## 1. Variables

### 1.1 Dynamic Typing

Declaring a variable in JavaScript can be done by using the var keyword followed by a name. Unlike C, C++, and Java, JavaScript is a dynamically-typed language; variables do not have explicit types associated with them. Instead, types are implicit.

var count;

Above we have declared a variable count. count has no associated type. We could initialize it to an integer, like so:

count = 10;

We could just as easily declare/initialize a string, double, or boolean:

var name = "John Doe";

var precision = 0.85;

var isPrecise = precision > 0.5; // true

### 1.2 Numbers

We can initialize variables that hold numbers using the syntax described in section 1.1. For example:

var age = 13; // in years

var height = 5.25; // in feet

Standard operations on numbers include adding, subtracting, multiplying, dividing, and modulus:

13 + 15; // 28

27 - 2; // 25

3 \* 5; // 15

40 / 20; // 2

10 % 6; // 4

Bitwise and (&), or (|), and xor (^) operators also exist. They function just as they would in C:

// 101 & 100 = 100

5 & 4; // 4

// 101 | 100 = 101

5 | 4; // 5

// 101 ^ 100 = 001

5 ^ 4; // 1

In contrast to C, there is no distinction between integer division and floating point division:

20 / 40; // 0.5

20.0 / 40.0; // 0.5

When it comes to overflow, know that JavaScript represents all numbers as 64 bit floats, resulting in a range of approximately 5e-324 to 1.7e308. Integers are considered reliable up to 15 digits, at which point they may need to be represented with digits after the decimal point. This implies that, after 15 digits, equality comparisons on integers are no longer safe, as they are effectively floats.

### 1.3 Strings

We may initialize strings using literal notation:

var phoneNumber = "(555) 555-5555";

When initializing strings, note that there is no difference between using single and double quotes. This also implies that JavaScript makes no distinctions between characters and strings. In other words, the following two statements are exactly equivalent:

var name = "John Doe";

// is the same as:

var name = 'John Doe';

We can use the standard + operator for concatenation, like so:

var firstName = 'John';

var lastName = 'Doe';

firstName + ' ' + lastName; // 'John Doe'

JavaScript provides a variety of useful string methods. Some notable ones include:

// extract substring from index 3 to index 6

name.substring(3, 6); // returns "n Do"

// extract character (actually returned as a string) at index 2;

name.charAt(2); // returns "h"

// searches for 'Doe' and returns its position or -1 otherwise;

name.indexOf('Doe'); // returns 5

// creates a new, uppercased version and returns it;

name.toUpperCase(); // returns 'JOHN DOE'; name is left unchanged

// creates a new, lowercased version and returns it;

name.toLowerCase(); // returns 'john doe'; name is left unchanged

// parse an integer out of the provided string

parseInt("3"); // returns 3

// parse a float out of the provided string

parseFloat("37.89"); // returns 37.89

// short hand for parseInt()/parseFloat()

+"3"; // 3

+"37.89"; // 37.89

Note that even though .charAt() seems to return a character, it is, in fact, returning a string.

A reference of other string methods can be found towards the bottom of [Mozilla Developer Network's (MDN's) documentation](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String).

### 1.4 Booleans

Booleans are true/false values, just as in C:

var isGeorgeTall = true;

var doPigsFly = false;

Common operators that return booleans include equality (== or ===), less than and greater than (< and >), and not (!):

3 < 4; // true

3 > 4; // false

== compares value whereas === compares type and value. ==, if used on different types, will coerce the types to be the same and then perform an equality comparison. ===, by contrast, will return false if used on different types:

3 == 3; // true

3 === 3; // true

3 == '3'; // true; '3' is coerced into an integer and compared with 3

3 === '3'; // false; types are different

Boolean expressions can be combined by using the and (&&) and or (||) operators, just as in C.

isGeorgeTall && doPigsFly; // false

isGeorgeTall || doPigsFly; // true

See chapter 2 for more details about conditionals.

### 1.5 Arrays

Array literals are defined just like in C, except by using square brackets instead of braces:

var numbers = [1, 2, 3];

var nouns = ["country", "tennis", "printer", "charger"];

Arrays need not contain variables of the same type. For example:

var numbersAndNouns = [1, "country", "tennis", 2];

To access an element in an array, use the familiar bracket notation:

numbers[0]; // 1

As shown by the above example, arrays are zero-indexed. To find the length of an array, use its length property:

nouns.length; // 4

Common array operations/methods include:

// adds 4 to the end of the array

numbers.push(4); // returns the new length of the array, which is 4

// numbers has now been mutated to [1, 2, 3, 4]

// removes the last element

numbers.pop(); // removes 4 and returns it

// numbers has now been mutated to [1, 2, 3]

// adds "object" to the start of the array

nouns.unshift("object"); // returns the new length of the array, which is 5

// nouns has now been mutated to ["object", "country", "tennis", "printer", "charger"]

// removes the first element

nouns.shift(); // removes "object" and returns it

// nouns has now been mutated to ["country", "tennis", "printer", "charger"]

// finds 2 in the array and returns its index; if not found, returns -1

numbers.indexOf(2); // returns 1

// concat one array to the end of another, returning a new, combined array

nouns.concat(numbers); // returns ["country", "tennis", "printer", "charger", 1, 2, 3]

// nouns and numbers remain unmodified

// sort an array; modify it directly and return the result

nouns.sort(); // returns ["charger", "country", "printer", "tennis"]

// numbers has now been mutated to ["charger", "country", "printer", "tennis"]

A reference of other array operations/methods may be found [on MDN](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array).

### 1.6 Undefined

Variables that are not explicitly assigned a value are, by default, initialized to the keyword undefined. For example, the two statements below are implicitly the same:

var name;

// is the same as:

var name = undefined;

### 1.7 Overriding

Defining the same variable again simply overrides the previous definition:

var name = "John";

var name = "Alex";

name; // "Alex"

In this case, name was originally defined as "John". The second definition of name in the same scope effectively acts as a reassignment. name thus becomes "Alex". See chapter 4 for more information about scopes.

### 1.8 Dynamic Typing Implications

Because types are dynamic, it should be noted that a variable of one type may always be assigned to a variable of another type. There is no distinction between what one variable can be assigned to and what another variable can be assigned to. For example, the following is completely valid:

var x = 10;

// this is OK, even though x was previously an integer

x = "some string";

This puts the burden on the programmer, then, to keep track of what types he/she is working with, as no variable is guaranteed to be of a specific type.

Additionally, dynamic typing, albeit easy on the programmer, leads to ambiguous functionality. For example, when doing a simple addition of two variables, JavaScript must consider their implicit types. Consider the following example:

var x = 20;

var y = 50;

var result = x + y; // 70

When both variables are integers, the addition is trivial. But what if we made x a string?

x = "20";

result = x + y; // "2050"

In this case, addition acts as concatenation, since x is actually a string. We can coerce a string to a numeric type by prepending a '+' operator, like so:

result = +x + y; // 70

An alternative method is to use the built-in parseInt/parseFloat function:

result = parseInt(x) + y;

Below are a few more examples of varying behavior:

x = true;

y = 1;

result = x + y; // 2

x = false;

result = x + y; // 1

x = "20";

result = x - y; // 19

In the last example, note that there is no subtraction operation defined for strings. As a result, JavaScript coerced x into a numeric type and then subtracted the integer y, resulting in 20 - 1 = 19.

Numerous other cases could be presented, but they all lead to one take-away point: it is the programmer's job to keep track of what variable types he/she is dealing with to not get unexpected behavior; dynamic typing is simply for ease-of-use.

## 2. Conditionals and Loops

### 2.1 Boolean Conditionals

Similar to most languages, JavaScript allows us to condition on true/false, or boolean, expressions. The simplest form of this is the if statement:

if (condition) {

// do something

}

condition, in this case, can be some boolean expression. For example:

if (3 == 3) {

// do something

}

As you can see, we use the standard == sign to denote comparison rather than assignment. Better yet, the === sign is more appropriate:

if (3 === 3) {

// do something

}

Not only does === compare equality, but it also compares types. This is the preferred comparison method to prevent unexpected behavior. To illustrate why, consider the following example:

var x = 3;

var y = "3";

if (x == y) {

x += y;

}

// x is now "33"

The == operator will coerce the string "3" into an integer and then compare it to the integer 3. These two are the same, and so the inner block of the if statement will be executed. This can lead to undesired functionality, as the statement x += y actually performs string concatenation (see section 1.8 for more details) rather than integer addition. Using the condition x === y would have averted this problem entirely, as the string "3" is not the same type as the integer 3.

### 2.2 Implicit Conditionals

But JavaScript, like C, allows us to put non-boolean values in place of condition. In these cases, JavaScript makes some assumptions about what values are considered to be true and what values are considered to be false. We call the former "truthy" values and the latter "falsy" values.

The following values are considered falsy:

* the integer zero (0)
* the empty string ("")
* null (null)
* not a number (NaN)
* undefined (undefined)
* false (false)

Everything else, by contrast, is truthy.

Some members of the JavaScript community prefer to use truthy/falsy comparisons; that is, they prefer this:

if (!name) {

// do something

}

instead of this:

if (name === undefined) {

// do something

}

Proponents of the former could argue that it is more readable; proponents of the latter could argue that it is more explicit and protects against type issues. Both are acceptable, but it all depends on context; if name is an empty string, the former may mistakenly execute the inner if block, and so one might prefer the latter; if name is known to be either a valid person's name or undefined, one might prefer the former. Use your best judgement.

### 2.3 Loops

Just as JavaScript and C share the if statement, they also share for/while loops, with exactly the same format. For example:

for (var i = 0; i < 10; i++) {

// do something 10 times

}

while (condition) {

// do something until condition no longer holds

}

Both these cases act exactly like they would in C; the former loops 10 times, incrementing i after each iteration. The latter loops while condition holds. As explained earlier, condition could either be an explicit boolean statement or an implicit truthy/falsy value.

While inside a loop, the following special statements may be issued:

* break; breaks out of the loop immediately
* continue; continues to the next iteration, skipping all inner-loop instructions below it

These statements mirror their C equivalents.

## 3. Functions

### 3.1 Named Functions

To define a function in JavaScript, use the function keyword followed by a name and list of arguments in parentheses, like so:

function add(a, b) {

return a + b;

}

Notice two main dissimilarities compared to C:

* Function arguments contain no type information. This is because JavaScript is dynamically-typed.
* Functions do not explicitly specify a return type. Any type may be returned by any function; there is no concept of differing prototypes. Functions that do not explicitly return a value return undefined by default.

We can then call this function as we would with a function in C:

add(5, 10); // returns 15

### 3.2 Function Overriding

If we define a function with the same name as a previously-defined function, it will override the previous function. This occurs regardless of whether the functions take in a differing number of arguments; remember, JavaScript does not make distinctions between functions based off their return value or their arguments. In other words, consider the following code:

function subtract(a, b) {

return a - b;

}

function subtract(a, b, c) {

return a - b - c;

}

subtract(10, 5, 2); // returns 3

The latter definition of subtract overrides the former.

### 3.3 Dropped/Default Arguments

What if we called subtract with an extra argument?

function subtract(a, b) {

return a - b;

}

subtract(10, 5, 2); // returns 5

Notice how the third argument to subtract is simply ignored. JavaScript will implicitly drop it (actually, this third argument is still accessible; see section 3.4).

What if we left off an argument entirely?

subtract(10); // returns NaN

All arguments that are not explicitly given a value are analogous to uninitialized variables. As such, they will be given a default value of undefined. Doing a subtraction using an undefined variable, by JavaScript standards, returns NaN (not a number).

### 3.4 Console Functions

Debugging in JavaScript primarily occurs through the console, which is available in most modern web browsers (Safari, Firefox, Chrome). To print data to the console, use the console.log() function, like so:

console.log('This is a log message.')

// you should see 'This is a log message.' in your console

We may pass in multiple arguments to console.log(); it will print all of them with padding in-between:

console.log('Some other message.', 5 \* 3);

// you should see 'Some other message. 15' in your console

Notice in that last example that we can also pass in numbers to console.log(). In fact, we can pass in any type and console.log() will display it in a human-readable format:

var evens = [2, 4];

evens.push(6);

console.log(evens);

// you should see '[2, 4, 6]' in your console

### 3.4 Variable Arguments

*Prerequisite: Please read section [TODO], which describes JavaScript objects, before learning about variable arguments.*

At times, it's useful to have a function that accepts a variable number of parameters. One common example of this in C is printf. To allow for this variability in JavaScript, a special arguments object is defined within each function call. This object contains a mapping of argument position to argument. For example, if one calls foo(10, 'bar', 11), arguments will be:

{

0: 10,

1: 'bar',

2: 11,

length: 3

}

Notice the extra length field, which contains the number of arguments passed in.

Recall that since we can pass in extra arguments or leave off arguments in any function call (see section 3.3), there is no need for any other special syntax to allow for variable argument processing. This can be seen with the simple example below:

function add() {

var sum = 0;

// add all arguments together

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

sum += arguments[i];

}

return sum;

}

As shown above, add() is variable-argument function that sums up all of its parameters and returns the result. To implement it, we simply process each argument in the arguments mapping, much like we would do for summing an array of numbers.

### 3.4 Anonymous Functions and Callbacks

Functions may also be anonymous. For example, we could omit the name of the subtract function we defined above, like so:

function(a, b) {

return a - b;

}

But we no longer have a reference to this function, meaning we can't call it. So why would we want to use anonymous functions? The answer comes in the form of parameters.

In C/C++, we can pass a function pointer as an argument to another function. An example of this is C's qsort(), which takes in an optional comparator function that defines the relationship (less than, greater than, or equal to) between two array elements. qsort() then calls this function to find a sorted ordering of the array elements.

In JavaScript, we can accomplish this same functionality easily with anonymous functions:

var numbers = [2, 1, 3];

numbers.sort(function(a, b) {

return b - a;

});

console.log(numbers); // logs '[3, 2, 1]'

The array sort() method takes in an optional comparator function. This function takes in two array elements, a and b, and provides the following functionality:

* If a should be placed after b in the sorted array, the function returns a positive number.
* If a and b can be interchanged in the sorted array (they are equal), this function returns 0.
* If a should be placed before b in the sorted array, this function returns a negative number.

This information is used by sort() to sort the array.

In the above example, we directly pass an anonymous function as this comparator. We don't plan to use this function anywhere else, so there's no point in giving it its own name. In fact, granting it a name will simply clutter the namespace. In turn, an anonymous function is ideal; only sort(), which actually needs the function, gets a reference to it.

Notice how we pass a function when calling sort() only to have sort() call it right back. Because of this, we refer to the function we pass in as a callback, since it is called back.

Note that we could have defined a named function and passed it in to sort() like so:

function compare(a, b) {

return b - a;

}

numbers.sort(compare);

As explained earlier, this exposes the compare function when nothing else needs a reference to it. This pollutes the namespace, making an anonymous function preferable.

Defining a function that takes in a callback is as we would expect:

function forEach(array, callback) {

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

callback(array[i], i);

}

}

forEach() takes in an array, array, and a callback, callback. It loops through array, calling callback for each element and index. Notice how the passing and calling of a function pointer is no different from regular parameter passing and regular function calls.

forEach() is a convenience function for iteration. In fact, in modern browsers, arrays have a forEach() method defined by default. See the [MDN documentation](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/forEach) for details.

### 3.5 Function Variables

Anonymous functions may be assigned to a variable to obtain a reference:

var subtract = function(a, b) {

return a - b;

};

Notice how we maintain a semicolon at the end of the function definition, as this is a variable assignment.

subtract is known as a function variable, as it is a variable that refers to a function. Note that if we assign a variable to a named function, the name of that function is replaced by the variable:

var subtract = function sub(a, b) {

return a - b;

};

console.log(subtract); // logs 'function […]'

// sub is no longer defined

*Prerequisite: Please read chapter 4 on scope and hoisting before continuing.*

Named functions are considered initialized in their whole scope, regardless of whether they are called before or after their definition. Function variables, by contrast, are treated like normal variables and are subject to hoisting (see section 4.2). For example:

square(10); // returns 100

function square(x) {

return x \* x;

}

Notice how we can call the named function square() before it is even defined. As long as the call to square() is somewhere within the scope of where square() is defined, this is legal.

With a function variable, however, this is not the case:

square(10); // error: undefined is not a function

var square = function(x) {

return x \* x;

};

When called, square is undefined due to hoisting. Only after the definition can we call square() safely:

var square = function(x) {

return x \* x;

};

square(10); // returns 100

## 4. Scope, Hoisting, and Closures

### 4.1 Scope

Unlike C, Java, and C++, a new scope in JavaScript is only created by functions. Consider the following code:

var name = 'John';

function logName() {

var name = 'Alex';

console.log(name);

}

console.log(name); // logs 'John'

logName(); // logs 'Alex'

logName() creates its own scope with its own name variable. The outer scope is unaware of this variable, and so the outer console.log() logs 'John'. In contrast, the inner console.log() prefers the local name variable over the one in the global scope, much in tune with C/C++/Java. As a result, it prints 'Alex'.

But what about the following code?

var name = 'John';

if (name.charAt(0) === 'J') {

var name = 'Alex';

console.log(name); // logs 'Alex'

}

console.log(name); // logs 'Alex'

The first console.log() logs 'Alex', as expected. But because if statements do not create new scopes, the outer name variable gets overridden by the inner name variable inside the if statement (see section 1.7 on variable overriding), as they are in the same scope. Thus, the second console.log() also logs 'Alex'.

Even more interesting is this case:

var name = 'John';

function logName() {

if (!name) {

var name = 'Alex';

}

console.log(name);

}

logName(); // logs 'Alex'

Inside logName(), it seems like name is already defined in the outer scope, so the if statement should not be entered. Nevertheless, the code still logs 'Alex'! Why? This brings us to the next topic: hoisting.

### 4.2 Hoisting

Variable declarations in JavaScript are always hoisted to the top of their scope. Consider the following code:

function averageCubes(numbers) {

var total = 0;

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

var cube = numbers[i] \* numbers[i] \* numbers[i];

total += cube;

}

return total / numbers.length;

}

averageCubes() takes in an array of numbers, numbers, and returns the average of their cubes. Since variables are hoisted to the top of their scope, the JavaScript engine actually interprets averageCubes as if it looked like this:

function averageCubes(numbers) {

var total;

var i;

var cube;

total = 0;

for (i = 0; i < numbers.length; i++) {

cube = numbers[i] \* numbers[i] \* numbers[i];

total += cube;

}

return total / numbers.length;

}

Notice how variable declarations have been moved to the top of their scope (recall that only functions create new scopes). Also notice how initializations stay where they are.

This behavior leads to unexpected functionality when variables are not explicitly declared at the beginning of their scopes. Consider the code presented at the end of section 4.1:

var name = 'John';

function logName() {

if (!name) {

var name = 'Alex';

}

console.log(name);

}

logName(); // logs 'Alex'

Why does this log 'Alex' as opposed to 'John'? Consider what this code looks like to the interpreter after hoisting in logName():

function logName() {

var name;

if (!name) {

name = 'Alex';

}

console.log(name);

}

Notice how the declaration of name in logName() is hoisted above the if statement. But recall from section 1.6 that variables not assigned a value default to undefined. Since undefined is a falsy value (see section 2.2), !name is actually true! Thus, name gets assigned to 'Alex' even though it appears that the if statement shouldn't be entered from the original code.

Because of such issues, it is advisable to declare all variables at the top of their respective scopes explicitly in the code. This way, any unexpected behavior due to hoisting is eliminated from the start.

### 4.3 Closures

*Prerequisite: Please read section [TODO], which describes JavaScript objects, before learning about variable arguments.*

A closure is a function that can access variables in its outer scope (lexical scope) even when invoked outside this scope. Consider the following code:

function Person() {

var name = "John Doe";

return {

getName: function() {

return name;

}

};

}

var john = Person();

console.log(john.getName()); // logs 'John Doe'

Even though name is defined in the context of the Person() function, it can be accessed when john.getName() is called. This is because getName() is a closure; it can access the variable name, which is in its outer scope, even when invoked outside of its outer scope.

All functions in JavaScript can act as closures; no special syntax is necessary. Closures are used often to create modules and emulate objects. The Person()example from above can be extended to show this:

function Person(name, age) {

return {

getName: function() {

return name;

},

setName: function(newName) {

name = newName;

},

getAge: function() {

return age;

},

setAge: function(newAge) {

age = newAge;

}

};

}

var george = Person("George", 30);

var jack = Person("Jack", 12);

console.log(george.getName()); // logs 'George'

jack.setAge(13);

console.log(jack.getAge()); // logs 13

Using the power of closures, we've effectively created a Person object that retains both a name and age. We can construct a Person by simply calling Person() with name and age parameters. Data is retained because the closures returned from Person() can continue to access name and age in their outer scope. In fact, the name and age variables associated with george and jack are guaranteed to live at least as long as the closures live.

Realize that even though JavaScript doesn't have visibility operators like Java, we are effectively creating private variables by using closures. name and age cannot be accessed directly; rather, they can only be modified by using the setName() and setAge() functions. In effect, then, we've achieved a form of encapsulation.

## 5. Objects

### 5.1 What are objects?

Both Java and Javascript represent nouns as objects (e.g. person, cup, laptop). Java uses classes as recipes for these objects. Javascript by contrast uses a simple collection of key-value pairs to represent these same objects.

### 5.2 Key Insertion

Objects can quite simply be instantiated using curly braces.

// Returns an instance of the Object constructor

// (we get to constructors in section 6.2). Both

// of the following two statements result in an

// empty object.

var a = {};

var a = new Object(); // We will discuss this form of

// Object instantiation in sections 6.1 & 6.2

There are two ways to add key-value pairs to the object.

var a = {};

// These two ways are functionally equivalent

a.property = "value";

a['property'] = "value";

The limitation of the former statement (the dot-operator access) is that some valid keys lead to interpreter errors. Consider the following code:

a.1property = "value"; // Leads to an interpreter error

a['1property'] = "value"; // Executes successfully

The dot-operator key instantiation only allows for string keys that start with certain initial characters (e.g. alphabetical or certain special characters such as $ but not numerical characters).

*Note: Any non-string values that are assigned as the key will be type coerced to its string representation*

### 5.3 Value Types

Although the keys that are used in Javascript objects must be strings, the values corresponding to those keys can be any representable object: function reference, arrays, other objects. However, every key-value pair in an object literal must be followed by a comma excluding the final pair.

var a = {

b: function() {

// Other statements

return;

},

c: [

"first",

"second",

"third"

],

d: "Some String"

}

### 5.5 The this operator

In JavaScript this always refers to the context a function is executed within. This is oftentimes the object within which the function is contained.

#### **Example: Modified from jblotus' Javascript blog**

console.log(this); // outputs window object

var myFunction = function() {

console.log(this);

};

myFunction(); //outputs window object

var newObject = {

myFunction: myFunction

};

newObject.myFunction(); //outputs newObject

Since myFunction() is a property of the global object, this is a reference to the global object, which is window. Consider the following example where the object newObject contains myFunction() now as a property. Since the surrounding object is now newObject, this refers to newObject - the "owner" of that function.

### Exceptions to *this* behavior

#### **1) When using *call* and *apply* functions, …**

When the call and apply functions are used, the programmer can decide what the context for this operator is. Consider the following extension of the previous example:

// As the first argument the programmer can determine the

// 'this' context to choose. Note: This property is widely

// used in functional programming implementations of the language

// (e.g. currying)

myFunction.call(newObject, 'foo', 'bar');

// Outputs newObject "foo" "bar"

myFunction.apply(newObject, ['foo', 'bar']);

// Outputs newObject "foo" "bar"

#### **2) When using the** new **operator …**

When using the new operator to create a new instance from a constructor, the keyword this refers to the newly generated instance (see section 6.2). Readers might find the following blog post on [helpful Javascript rules of thumb by Henry Zhu](http://henrycode.tumblr.com/post/37627169791/javascript-clarifying-the-keyword-this) to be informative:

### 5.4 The DOM Global Object

Within the browser, the global scope of the objects held within the browser is represented by the window object. By default the context of this contained within the global scope is the window object.

// Assuming that this statement is not contained within a function.

console.log(this); // Returns window object

var a = {

b: function() {

// Returns the object 'a'

console.log(this);

}

}

## 6. Inheritance Model

### 6.1 The new operator

Before discussing Javascript's inheritance model, we need to discuss object generation from constructors via the new operator, a concept that is often greatly misunderstood in the Javascript programming community. Essentially, the new operator causes a constructor function to be called with the operator this bound to a newly created Object whose prototype is that function's prototype property. Let's dissect a practical usage of new.

// This is a constructor function that is used to define initial

// properties on a newly generated instance. We will go into more

// depth in section 6.2

var Circle = function(radius) {

this.radius = radius;

// next line is implicit, added for illustration only

// this.\_\_proto\_\_ = Circle.prototype;

}

// . . . Other inheritance code.

// The new operator binds the 'this' operator in the Circle

// constructor to the newly created instance of the object 'a'.

// Also the new operator implicitly sets the actual prototype of the

// object 'a' to the prototype object returned by Circle's prototype

// property. Therefore, this.radius sets a property of 'radius'

// on object 'a'.

var a = new Circle(3);

// Returns 3 since the 'new' operator bound 'this' to the

// instance object 'a'.

### 6.2 Constructor Functions

Constructors in Javascript in certain ways mirror those in Java in that when a new object is instantiated, this function is immediately invoked. However, in Javascript the user directly invokes the constructor, preceded by the new operator (as described in section 6.1), to create an instance of the desired object. Let's revisit the previous example in section 6.1 and examine the constructor in more detail.

function Circle(radius) {

// The context of \*this\* (as described in section 5.2)

// relates to the newly created instance.

this.radius = radius;

// The \*new\* operator implicitly adds this modification to

// the instance object's prototype chain. The line has been

// added in simply for clarity of understanding.

// this.\_\_proto\_\_ = Circle.prototype;

}

Now that we have a solid understanding of the new operator and constructor functions, we are well positioned to discuss Javascript's inheritance model called the prototype chain. This inheritance model differs greatly from the traditional inheritance model of traditional languages such as C / C++ / Java.

### 6.3 Prototype

But let's take a step back and define what a prototype exactly is. Simply put, a prototype is a Javascript object from which other objects inherit properties. Consider an object A, which has a prototype of object B from which it inherits properties. This object B in turn has a prototype from object C. And so on. This repeated chain of inheriting object properties from a prototype continues until it hits the fundamental building block of Javascript objects – the global Object which has no prototype.

(A → B → C → … → Object)

### 6.4 Prototype Property

Perhaps one of the most misunderstood concepts in Javascript is the distinction between the prototype *property* and a certain object's actual prototype as defined in the previous section. The prototype property does not return the prototype of the object that calls it but rather defines an object that will serve as the prototype for all the instances of that object (this will be explained in section 6.5). This prototype property is only contained in function objects. Consider the following Javascript.

function Man(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

// We will discuss the new operator in more depth later

// but it basically creates an instance of the function

// Man using Man's defined prototype properties.

var man = new Man();

// The '\_\_proto\_\_' property is contained within all browsers

// , except browsers < IE9 and returns the actual prototype object

// in the inheritance tree. The 'prototype' property defined on the

// 'Man' function does not return the prototype of Man but the

// prototype of the instances of Man - in this case 'man'.

console.log(man.\_\_proto\_\_ == Man.prototype); // Returns true

// This code serves to highlight the point that the \_\_proto\_\_ of

// Man function is \*not\* the same as the object returned by the

// prototype property.

console.log(Man.\_\_proto\_\_ == Man.prototype); // Returns false

### 6.5 Prototype Property in Action

So why is this prototype property so important to understanding the Javascript inheritance pattern? By using this property, we are able to define properties of instances of that constructor function.

#### **Example 1: Following the same example from above:**

// We can describe this as the constructor of the function Man.

// This function will always be called when a new instance of that

// function is created via 'new'.

function Man(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

// This defines the function 'sayName' as a property of the prototype

// property. This means that all instances of Man will have an actual

// prototype object with 'sayName' defined as a property.

Man.prototype.sayName = function() {

console.log(this.firstName + " " + this.lastName);

};

var johnDoe = new Man("John", "Doe");

// These next two executed statements should log the same results.

// Remember from the previous section that man.\_\_proto\_\_ is equivalent

// to Man.prototype.

johnDoe.sayName(); // This returns 'John Doe'

// Logging out the Man prototype property in Google Chrome, we notice

// the following properties:

// -> Man

// -> constructor

// -> sayName

// -> \_\_proto\_\_

// We now can see that sayName has been dynamically added to the Man

// object.

console.log(Man.prototype);

#### **Example 2: by Angus Croll, a less contrived example**

// Constructor. This is returned as new object and its

// internal \_\_proto\_\_ property will be set to the constructor's

// default prototype property

var Circle = function(radius) {

this.radius = radius;

// next line is implicit, added for illustration only

// this.\_\_proto\_\_ = Circle.prototype;

}

// Augment Circle's default prototype property thereby augmenting the

// prototype of each generated instance

Circle.prototype.area = function() {

return Math.PI \* this.radius \* this.radius;

};

// Create two instances of a circle and make each leverage

// the common prototype

var a = new Circle(3), b = new Circle(4);

a.area().toFixed(2); // 28.27

b.area().toFixed(2); // 50.27

console.log(a.radius);