


Branch: master [netcom\\_advanced\\_python / day-2 / 3-reintroduction-to-javascript.md](#) Find file Copy path

 **t-0-m-1-3** initial commit 6b7420b 17 hours ago

1 contributor

819 lines (694 sloc) 28.1 KB Raw Blame History

## A re-introduction to JavaScript

- Because JavaScript is notorious for being the world's most misunderstood programming language.
- It is often derided as being a toy, but beneath its layer of deceptive simplicity, powerful language features await.
- JavaScript is now used by an incredible number of high-profile applications
- JavaScript was created in 1995 by Brendan Eich while he was an engineer at Netscape.
- JavaScript was first released with Netscape 2 early in 1996.
  - It was originally going to be called `LiveScript`, but it was renamed in an ill-fated marketing decision that attempted to capitalize on the popularity of Sun Microsystem's Java language — despite the two having very little in common.
- Microsoft released JScript with Internet Explorer 3. It was a mostly-compatible JavaScript work-alike.
- Several months after that, Netscape submitted JavaScript to Ecma International, a European standards organization
- The fourth edition was abandoned, due to political differences concerning language complexity. Many parts of the fourth edition formed the basis for ECMAScript edition 5, published in December of 2009, and for the 6th major edition of the standard, published in June of 2015.
- JavaScript language has no concept of input or output. It is designed to run as a scripting language in a host environment, and it is up to the host environment to provide mechanisms for communicating with the outside world
  - most common host environment is the browser, but JavaScript interpreters can also be found in a huge list of other places

### Overview

\*JavaScript is a multi-paradigm, dynamic language with types and operators, standard built-in objects, and methods.

- Its syntax is based on the Java and C languages — many structures from those languages apply to JavaScript as well.
- JavaScript supports object-oriented programming with object prototypes, instead of classes (see more about prototypical inheritance and ES2015 classes).
- JavaScript also supports functional programming — because they are objects, functions may be stored in variables and passed around like any other object.

## JavaScript Types

- Number
- String
- Boolean
- Symbol (new in ES2015)
- Object
  - Function
  - Array
  - Date
  - RegExp
- null
- undefined

### Number

- Numbers in JavaScript are "double-precision 64-bit format IEEE 754 values", according to the spec. This has some interesting consequences.
- There's no such thing as an integer in JavaScript

### Watch out for stuff life

```
0.1 + 0.2 == 0.30000000000000004;
```

- integer values are treated as 32-bit ints, and some implementations even store it that way until they are asked to perform an instruction
- standard arithmetic operators are supported, including addition, subtraction, modulus (or remainder) arithmetic
- built-in `Math` object

```
Math.sin(3.5);  
var circumference = 2 * Math.PI * r;
```

- convert a string to an integer using the built-in `parseInt()` function

```
parseInt('123', 10); // 123  
parseInt('010', 10); // 10
```

- parse floating point numbers using the built-in `parseFloat()` function. Unlike `parseInt()`, `parseFloat()` always uses base 10
- can also use the unary operator to convert values to numbers

```
+ '42'; // 42  
+ '010'; //10  
+ '0x10'; //16
```

- `NaN` is returned in the string is non-numeric

```
parseInt('hello', 10); //NaN
```

- `NaN` in any math operation will result in `NaN`
- Test for `NaN` using `isNaN()` built-in

```
isNaN(NaN); //true
```

- JavaScript also supports `Infinity` and `-Infinity`

```
1 / 0; //Infinity
```

- Test for that using the built-in `isFinite()`

```
isFinite(1 / 0); // false  
isFinite(-Infinity); //false
```

## Strings

- Strings in JavaScript are sequences of Unicode characters
- They are sequences of UTF-16 code units; each code unit is represented by a 16-bit number.
- Each Unicode character is represented by either 1 or 2 code units.
- find the length of a string by accessing its `length` property

```
'hello'.length; //5
```

- Strings are objects, with methods on them too

```
'hello'.charAt(0); // "h"  
'hello, world'.replace('world', 'mars'); // "hello, mars"  
'hello'.toUpperCase(); // "HELLO"
```

## Other Types

- JavaScript distinguishes between `null`, which is a value that indicates a deliberate non-value (and is only accessible through the `null` keyword), and `undefined`, which is a value of type `undefined` that indicates an uninitialized variable
- JavaScript has a boolean type, with possible values `true` and `false` (both of which are keywords.) Any value can be converted to a boolean according to the following rules
  - `false`, `0`, empty strings `""`, `NaN`, `null`, and `undefined` all become `false`
  - everything else is `true`
  - you can convert explicitly using the `Boolean()`

```
Boolean(''); //false  
Boolean(''); //true
```

## Variables

- Three ways to declare a variable: `var`, `const`, `let`

- `let` allows you to declare block-level variables

```
let a;
let name = 'Simon';
```

- `let` has scope

```
// myLetVariable is *not* visible out here

for (let myLetVariable = 0; myLetVariable < 5; myLetVariable++) {
  // myLetVariable is only visible in here
}

// myLetVariable is *not* visible out here
```

- `const` allows you to declare variables whose values are never intended to change

```
const Pi = 3.14; // variable Pi is set
Pi = 1; // will throw an error because you cannot change a constant variable.
```

- `var` is the most common declarative keyword, traditional way

```
var a;
var name = 'Simon';
// and the scope example
// myVarVariable *is* visible out here

for (var myVarVariable = 0; myVarVariable < 5; myVarVariable++) {
  // myVarVariable is visible to the whole function
}

// myVarVariable *is* visible out here
```

- If you declare a variable without assigning any value to it, its type is `undefined`
- An important difference between JavaScript and other languages like Java is that in JavaScript, blocks do not have scope; only functions have a scope.
  - So if a variable is defined using `var` in a compound statement (for example inside an `if` control structure), it will be visible to the entire function

## Operators

- JavaScript's numeric operators are `+`, `-`, `*`, `/` and `%`
- Values are assigned using `=`, and there are also compound assignment statements such as `+=` and `-=`. These extend out to `x = x operator y`.

```
x += 5;
x = x + 5;
```

- can use `++` and `--` to increment or decrement
- `+` also does string concatenation

```
'hello' + 'world'; // "hello world"
```

- Adding a string to a number and everything is converted to a string

```
'3' + 4 + 5; //345
3 + 4 + '5'; //75
```

- Adding an empty string "" is a useful way to convert something to a string
- **Comparisons** in JS are made using <, >, <=, >= . They work for both strings and numbers
- != and !== as well as bitwise operators
- == performs coercion of different types

```
123 == '123'; // true
1 == true; // true
// avoid type conversion with
123 === '123'; // false
1 === true; // false
```

## Control Structures

- JavaScript has a similar set of control structures to other languages in the C family.
- Conditional statements are supported by if and else ; you can chain them together

```
var name = 'kittens';
if (name == 'puppies') {
  name += ' woof';
} else if (name == 'kittens') {
  name += ' meow';
} else {
  name += '!';
}
name == 'kittens meow';
```

- JS has while loops and do-while loops too

```
while (true) {
  // an infinite loop!
}
//executes at least once
var input;
do {
  input = get_input();
} while (inputIsValid(input));
```

- JS for loop is the same structure as c and Java

```
for (var i = 0; i < 5; i++) {
  // Will execute 5 times
}
```

- for...of :

```
for (let value of array) {
  // do something
}
```

- for...in :

```
for (let property in object) {
  // do something with object property
}
```

- `&&` and `||` will execute the second operand depending on the first -> useful for checking for null objects before accessing an attribute

```
var name = o && o.getName();
```

- or caching a value, if falsy values are invalid

```
var name = cachedName || (cachedName = getName());
```

- JS also has a ternary operator for conditionals

```
var allowed = (age > 18) ? 'yes' : 'no';
```

- `switch` statement use for multiple branchings

```
switch (action) {
  case 'draw':
    drawIt();
    break;
  case 'eat':
    eatIt();
    break;
  default:
    doNothing();
}
```

- Not adding a `break` statement will cause the execution to fall through to the next level (this isn't what you want)

```
switch (a) {
  case 1: // fallthrough
  case 2:
    eatIt();
    break;
  default:
    doNothing();
}
```

## Objects

JavaScript objects can be thought of as simple collections of name-value pairs. As such, they are similar to:

- Dictionaries in Python.
- Hashes in Perl and Ruby.
- Hash tables in C and C++.
- HashMaps in Java.
- Associative arrays in PHP.

The fact that this data structure is so widely used is a testament to its versatility

- everything (bar core types) in JavaScript is an object, any JavaScript program naturally involves a great deal of hash table lookups
- The "name" part is a JavaScript string, while the value can be any JavaScript value — including more objects. This allows you to build data structures of arbitrary complexity.
- There are two basic ways to create an empty object:

```
var obj = new Object();
```

And:

```
var obj = {};
```

- These are semantically equivalent;
- the second is called **object literal** syntax and is more convenient. This syntax is also the core of JSON format and should be preferred at all times.
- Object literal syntax can be used to initialize an object in its entirety:

```
var obj = {  
  name: 'Carrot',  
  for: 'Max', // 'for' is a reserved word, use '_for' instead.  
  details: {  
    color: 'orange',  
    size: 12  
  }  
};
```

- Attribute access can be chained together:

```
obj.details.color; // orange  
obj['details']['size']; // 12
```

- Creating an object prototype

```
function Person(name, age) {  
  this.name = name;  
  this.age = age;  
}  
  
// Define an object  
var you = new Person('Tom', 31);  
// We are creating a new person named "Tom" aged 31.
```

- Once you create the object, it's properties can be accessed using

```
// dot notation  
obj.name = 'Simon';  
var name = obj.name;  
  
// bracket notation  
obj['name'] = 'Simon';  
var name = obj['name'];  
  
// can use a variable to define a key  
var user = prompt('what is your key?')  
obj[user] = prompt('what is its value?')
```

- These are also semantically equivalent.
- The second method has the advantage that the name of the property is provided as a string, which means it can be calculated at run-time.
  - However, using this method prevents some JavaScript engine and minifier optimizations being applied

## Arrays

- Arrays in JavaScript are actually a special type of object.
- They work very much like regular objects (numerical properties can naturally be accessed only using [] syntax) but they have one magic property called 'length'.
  - This is always one more than the highest index in the array.
- One way of creating arrays is as follows:

```
var a = new Array();
a[0] = 'dog';
a[1] = 'cat';
a[2] = 'hen';
a.length; // 3
```

- A more convenient notation is to use an array literal:

```
var a = ['dog', 'cat', 'hen'];
a.length; // 3
```

Note that `array.length` isn't necessarily the number of items in the array. Consider the following:

```
var a = ['dog', 'cat', 'hen'];
a[100] = 'fox';
a.length; // 101
```

### Remember — the length of the array is one more than the highest index.

- If you query a non-existent array index, you'll get a value of undefined in return:

```
typeof a[90]; // undefined
```

- If you take the above about [] and length into account, you can iterate over an array using the following for loop:

```
for (var i = 0; i < a.length; i++) {
  // Do something with a[i]
}
```

####ES2015 introduced the more concise `for...of` loop for iterable objects such as arrays:

```
for (const currentValue of a) {
  // Do something with currentValue
}
```

- You could also iterate over an array using a `for...in` loop, however this does not iterate over the array elements, but the array indices.
  - If someone added new properties to `Array.prototype`, they would also be iterated over by such a loop.



Therefore this loop type is not recommended for arrays.

- Another way of iterating over an array that was added with ECMAScript 5 is `forEach()` :

```
['dog', 'cat', 'hen'].forEach(function(currentValue, index, array) {  
  // Do something with currentValue or array[index]  
});
```

- If you want to append an item to an array simply do it like this:

```
a.push(item);
```

Method Name	Description
<code>a.toString()</code>	returns a string with the <code>toString()</code> of each element separated by commas
<code>a.toLocaleString()</code>	Returns a string with the <code>toLocaleString()</code> of each element separated by commas.
<code>a.concat(item1[, item2[, ...[, itemN]])</code>	Returns a new array with the items added on to it.
<code>a.join(sep)</code>	Converts the array to a string — with values delimited by the <code>sep</code> param
<code>a.pop()</code>	Removes and returns the last item.
<code>a.push(item1, ..., itemN)</code>	Appends items to the end of the array.
<code>a.reverse()</code>	Reverses the array.
<code>a.shift()</code>	Removes and returns the first item.
<code>a.slice(start[, end])</code>	Returns a sub-array.
<code>a.sort([cmpfn])</code>	Takes an optional comparison function.
<code>a.splice(start, delcount[, item1[, ...[, itemN]])</code>	Lets you modify an array by deleting a section and replacing it with more items.
<code>a.unshift(item1[, item2[, ...[, itemN]])</code>	Prepends items to the start of the array.

Functions

- Along with objects, functions are the core component in understanding JavaScript.

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

- This demonstrates a basic function. A JavaScript function can take 0 or more named parameters.
- The function body can contain as many statements as you like and can declare its own variables which are local to that function.
- The return statement can be used to return a value at any time, terminating the function.
- If no return statement is used (or an empty return with no value), JavaScript returns undefined.

- The named parameters turn out to be more like guidelines than anything else. You can call a function without passing the parameters it expects, in which case they will be set to undefined.

```
add(); // NaN
// You can't perform addition on undefined
```

You can also pass in more arguments than the function is expecting:

```
add(2, 3, 4); // 5
// added the first two; 4 was ignored
```

- Functions have access to an additional variable inside their body called `arguments`, which is an array-like object holding all of the values passed to the function.

```
function add() {
  var sum = 0;
  for (var i = 0, j = arguments.length; i < j; i++) {
    sum += arguments[i];
  }
  return sum;
}
```

```
add(2, 3, 4, 5); // 14
```

- an averaging function:

```
function avg() {
  var sum = 0;
  for (var i = 0, j = arguments.length; i < j; i++) {
    sum += arguments[i];
  }
  return sum / arguments.length;
}
```

```
avg(2, 3, 4, 5); // 3.5
```

- To reduce this code a bit more we can look at substituting the use of the `arguments` array through Rest parameter syntax.
- In this way, we can pass in any number of arguments into the function while keeping our code minimal.
- The rest parameter operator is used in function parameter lists with the format: `...variable` and it will include within that variable the entire list of uncaptured arguments that the function was called with.

```
function avg(...args) {
  var sum = 0;
  for (let value of args) {
    sum += value;
  }
  return sum / args.length;
}
```

```
avg(2, 3, 4, 5); // 3.5
```

In the above code, the variable `args` holds all the values that were passed into the function.

- JavaScript lets you call a function with an arbitrary array of arguments, using the `apply()` method of any function object.

```
avg.apply(null, [2, 3, 4, 5]); // 3.5
```

- The second argument to `apply()` is the array to use as arguments;
- You can achieve the same result using the spread operator in the function call.

For instance: `avg(...numbers)`

- JavaScript lets you create anonymous functions.

```
var avg = function() {
  var sum = 0;
  for (var i = 0, j = arguments.length; i < j; i++) {
    sum += arguments[i];
  }
  return sum / arguments.length;
};
```

- It's extremely powerful, as it lets you put a full function definition anywhere that you would normally put an expression.
  - This enables all sorts of clever tricks. Here's a way of "hiding" some local variables — like block scope in C:

```
var a = 1;
var b = 2;
```

```
(function() {
  var b = 3;
  a += b;
})();
```

```
a; // 4
b; // 2
```

- JavaScript allows you to call functions recursively.
  - This is particularly useful for dealing with tree structures, such as those found in the browser DOM.

```
function countChars(elm) {
  if (elm.nodeType == 3) { // TEXT_NODE
    return elm.nodeValue.length;
  }
  var count = 0;
  for (var i = 0, child; child = elm.childNodes[i]; i++) {
    count += countChars(child);
  }
  return count;
}
```

**This highlights a potential problem with anonymous functions: how do you call them recursively if they don't have a name?**

- JavaScript lets you name function expressions for this. You can use named **IIFEs (Immediately Invoked Function Expressions)**

```
var charsInBody = (function counter(elm) {
  if (elm.nodeType == 3) { // TEXT_NODE
    return elm.nodeValue.length;
  }
  var count = 0;
```

```

    for (var i = 0, child; child = elm.childNodes[i]; i++) {
        count += counter(child);
    }
    return count;
})(document.body);

```

- The name provided to a function expression as above is only available to the function's own scope.
- This allows more optimizations to be done by the engine and results in more readable code.
- The name also shows up in the debugger and some stack traces, which can save you time when debugging.

## Custom objects

In classic Object Oriented Programming, objects are collections of data and methods that operate on that data.

- JavaScript is a prototype-based language that contains no class statement, as you'd find in C++ or Java (this is sometimes confusing for programmers accustomed to languages with a class statement).
- JavaScript uses functions as classes.

```

function makePerson(first, last) {
    return {
        first: first,
        last: last
    };
}
function personFullName(person) {
    return person.first + ' ' + person.last;
}
function personFullNameReversed(person) {
    return person.last + ', ' + person.first;
}

var s = makePerson('Simon', 'Willison');
personFullName(s); // "Simon Willison"
personFullNameReversed(s); // "Willison, Simon"

```

- This works, but it's pretty ugly. You end up with dozens of functions in your global namespace.
  - What we really need is a way to attach a function to an object. Since functions are objects, this is easy:

```

function makePerson(first, last) {
    return {
        first: first,
        last: last,
        fullName: function() {
            return this.first + ' ' + this.last;
        },
        fullNameReversed: function() {
            return this.last + ', ' + this.first;
        }
    };
}

var s = makePerson('Simon', 'Willison');
s.fullName(); // "Simon Willison"
s.fullNameReversed(); // "Willison, Simon"

```

Note on the `this` keyword. Used inside a function, `this` refers to the current object. What that actually means is specified by the way in which you called that function. If you called it using dot notation or bracket notation on an object, *that* object becomes `this`. If dot notation wasn't used for the call, `this` refers to the `global` object.

**Note that this is a frequent cause of mistakes. For example:**

```
var s = makePerson('Simon', 'Willison');
var fullName = s.fullName;
fullName(); // undefined undefined
```

When we call `fullName()` alone, without using `s.fullName()`, `this` is bound to the `global` object.

We can take advantage of the `this` keyword to improve our `makePerson` function:

```
function Person(first, last) {
  this.first = first;
  this.last = last;
  this.fullName = function() {
    return this.first + ' ' + this.last;
  };
  this.fullNameReversed = function() {
    return this.last + ', ' + this.first;
  };
}
var s = new Person('Simon', 'Willison');
```

- Another keyword: `new`. `new` is strongly related to `this`. It creates a brand new empty object, and then calls the function specified, with `this` set to that new object.
  - Notice though that the function specified with `this` does not return a value but merely modifies the `this` object.
  - It's `new` that returns the `this` object to the calling site.
  - Functions that are designed to be called by `new` are called **constructor functions**. Common practice is to capitalize these functions as a reminder to call them with `new`.

The improved function still has the same pitfall with calling `fullName()` alone.

Every time we create a person object we are creating two brand new function objects within it — wouldn't it be better if this code was shared?

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

- We are creating the method functions only once, and assigning references to them inside the constructor. Can we do any better than that? The answer is yes:

```
function Person(first, last) {
  this.first = first;
  this.last = last;
```

```

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

```

- `Person.prototype` is an object shared by all instances of `Person`.
- It forms part of a lookup chain (that has a special name, "prototype chain"): any time you attempt to access a property of `Person` that isn't set, JavaScript will check `Person.prototype` to see if that property exists there instead.
  - As a result, anything assigned to `Person.prototype` becomes available to all instances of that constructor via the `this` object.

This is an incredibly powerful tool. JavaScript lets you modify something's prototype at any time in your program, which means you can add extra methods to existing objects at runtime:

```

var s = new Person('Simon', 'Willison');
s.firstNameCaps(); // TypeError on line 1: s.firstNameCaps is not a function

Person.prototype.firstNameCaps = function() {
  return this.first.toUpperCase();
};
s.firstNameCaps(); // "SIMON"

```

- Can also add things to the prototype of built-in JavaScript objects:

```

var s = 'Simon';
s.reversed(); // TypeError on line 1: s.reversed is not a function

String.prototype.reversed = function() {
  var r = '';
  for (var i = this.length - 1; i >= 0; i--) {
    r += this[i];
  }
  return r;
};

s.reversed(); // nomiS

```

- Our new method even works on string literals!

```
'This can now be reversed'.reversed(); // desrever eb won nac sihT
```

- The prototype forms part of a chain. The root of that chain is `Object.prototype`, whose methods include `toString()` — it is this method that is called when you try to represent an object as a string. This is useful for debugging our `Person` objects:

```

var s = new Person('Simon', 'Willison');
s.toString(); // [object Object]

Person.prototype.toString = function() {
  return '<Person: ' + this.fullName() + '>';
};

s.toString(); // "<Person: Simon Willison>"

```

- The first argument to `apply()` is the object that should be treated as `this`. For example, here's a trivial

implementation of new:

```
function trivialNew(constructor, ...args) {
  var o = {}; // Create an object
  constructor.apply(o, args);
  return o;
}
```

- In this snippet, `...args` (including the ellipsis) is called the **"rest arguments"** — as the name implies, this contains the rest of the arguments.

## Calling

```
var bill = trivialNew(Person, 'William', 'Orange');
```

- Almost equivalent to

```
var bill = new Person('William', 'Orange');
```

- `apply()` has a sister function named `call`, which again lets you set `this` but takes an expanded argument list as opposed to an array.

```
function lastNameCaps() {
  return this.last.toUpperCase();
}
var s = new Person('Simon', 'Willison');
lastNameCaps.call(s);
// Is the same as:
s.lastNameCaps = lastNameCaps;
s.lastNameCaps(); // WILLISON
```

## Inner functions

- JavaScript function declarations are allowed inside other functions.
- An important detail of nested functions in JavaScript is that they can access variables in their parent function's scope:

```
function parentFunc() {
  var a = 1;

  function nestedFunc() {
    var b = 4; // parentFunc can't use this
    return a + b;
  }
  return nestedFunc(); // 5
}
```

- If a called function relies on one or two other functions that are not useful to any other part of your code, you can nest those utility functions inside it.
- This is also a great counter to the lure of global variables.
- When writing complex code it is often tempting to use global variables to share values between multiple functions — which leads to code that is hard to maintain.
- Nested functions can share variables in their parent, so you can use that mechanism to couple functions together when it makes sense without polluting your global namespace. This technique should be used with

caution, but it's a useful ability to have.

## Closures

What does this do?

```
function makeAdder(a) {  
  return function(b) {  
    return a + b;  
  };  
}  
var add5 = makeAdder(5);  
var add20 = makeAdder(20);  
add5(6); // ?  
add20(7); // ?
```

The name of the `makeAdder()` function should give it away: it creates new 'adder' functions, each of which, when called with one argument, adds it to the argument that it was created with.

- Pretty much the same as was happening with the inner functions earlier on: a function defined inside another function has access to the outer function's variables.
- The only difference here is that the outer function has returned, and hence common sense would seem to dictate that its local variables no longer exist.

```
add5(6); // returns 11  
add20(7); // returns 27
```

Here's what's actually happening. Whenever JavaScript executes a function, a 'scope' object is created to hold the local variables created within that function.

It is initialized with any variables passed in as function parameters. This is similar to the global object that all global variables and functions live in, but with a couple of important differences: firstly, a brand new scope object is created every time a function starts executing, and secondly, unlike the global object (which is accessible as `this` and in browsers as `window`) these scope objects cannot be directly accessed from your JavaScript code.

So when `makeAdder()` is called, a scope object is created with one property: `a`, which is the argument passed to the `makeAdder()` function. `makeAdder()` then returns a newly created function.

Normally JavaScript's garbage collector would clean up the scope object created for `makeAdder()` at this point, but the returned function maintains a reference back to that scope object.

As a result, the scope object will not be garbage-collected until there are no more references to the function object that `makeAdder()` returned.

Scope objects form a chain called the scope chain, similar to the prototype chain used by JavaScript's object system.

A **closure** is the combination of a function and the scope object in which it was created.

Closures let you save state — as such, they can often be used in place of objects.