Programming language Concepts

- Keyword List

Keywords are tokens that have special meaning in JavaScript: `break`, `case`, `catch`, `continue`, `debugger`, `default`, `delete`, `do`, `else`, `finally`, `for`, `function`, `if`, `in`, `instanceof`, `new`, `return`, `switch`, `this`, `throw`, `try`, `typeof`, `var`, `void`, `while`, and `with`.

Future reserved words are tokens that may become keywords in a future revision of ECMAScript: `class`, `const`, `enum`, `export`, `extends`, `import`, and `super`. Some future reserved words only apply in strict mode: `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static`, and `yield`.

The *null literal* is, simply, `null`.

There are two *boolean literals*: `true` and `false`.

- Declarations

`var`

Declares a variable, optionally initializing it to a value.

`let`

Declares a block scope local variable, optionally initializing it to a value.

`const`

Declares a read-only named constant.

- Declaring variables

- With the keyword `var`. For example, `var x = 42`. This syntax can be used to declare both local and global variables.

- By simply assigning it a value. For example, `x = 42`. This always declares a global variable. It generates a strict JavaScript warning. You shouldn't use this variant.
- With the keyword `let`. For example, `let y = 13`. This syntax can be used to declare a block scope local variable.

- Evaluating variables

```
var a;

console.log("The value of a is " + a); // The value of a is undefined

console.log("The value of b is " + b); // Uncaught ReferenceError: b
is not defined

console.log("The value of c is " + c); // The value of c is undefined

var c;

console.log("The value of x is " + x); // Uncaught ReferenceError: x
is not defined

let x;
```

You can use `undefined` to determine whether a variable has a value. In the following code, the variable `input` is not assigned a value, and the `if` statement evaluates to `true`.

```
var input;
if(input === undefined){
  doThis();
} else {
  doThat();
}
```

- Data Types

- Number. Numbers in JavaScript are stored as 64-bit floats, and can be manipulated using the built-in operators
- String. Strings are always declared through quotations
- Boolean. Booleans are the logical values, either true or false
- Symbol
- Object
- undefined
- null

Data structures and types

Data types

The latest ECMAScript standard defines seven data types:

- Six data types that are primitives:
  - Boolean. `true` and `false`.
  - null. A special keyword denoting a null value. Because JavaScript is case-sensitive, `null` is not the same as `Null`, `NULL`, or any other variant.
  - undefined. A top-level property whose value is undefined.
  - Number. `42` or `3.14159`.
  - String. "Howdy"
  - Symbol (new in ECMAScript 2015). A data type whose instances are unique and immutable.
- and Object

Although these data types are a relatively small amount, they enable you to perform useful functions with your applications. `Objects` and `functions` are the other fundamental elements in the language. You can think of objects as named containers for values, and functions as procedures that your application can perform.

JavaScript is a dynamically typed language. That means you don't have to specify the data type of a variable when you declare it, and data types are converted automatically as needed during script execution. So, for example, you could define a variable as follows:

```
var answer = 42;
```

And later, you could assign the same variable a string value, for example:

```
answer = "Thanks for all the fish...";
```

Because JavaScript is dynamically typed, this assignment does not cause an error message.

In expressions involving numeric and string values with the + operator, JavaScript converts numeric values to strings. For example, consider the following statements:

```
x = "The answer is " + 42 // "The answer is 42"
```

```
y = 42 + " is the answer" // "42 is the answer"
```

In statements involving other operators, JavaScript does not convert numeric values to strings. For example:

```
"37" - 7 // 30
```

```
"37" + 7 // "377"
```

Converting strings to numbers

In the case that a value representing a number is in memory as a string, there are methods for conversion.

- `parseInt()`
- `parseFloat()`

`parseInt` will only return whole numbers, so its use is diminished for decimals. Additionally, a best practice for `parseInt` is to always include the radix parameter. The radix parameter is used to specify which numerical system is to be used.

An alternative method of retrieving a number from a string is with the `+` (unary plus) operator:

```
"1.1" + "1.1" = "1.11.1"
```

```
(+"1.1") + (+"1.1") = 2.2
// Note: the parentheses are added for clarity, not required.
```

Literals

You use literals to represent values in JavaScript. These are fixed values, not variables, that you *literally* provide in your script. This section describes the following types of literals:

- Array literals
- Boolean literals
- Floating-point literals
- Integers
- Object literals
- RegExp literals
- String literals

Array literals

An array literal is a list of zero or more expressions, each of which represents an array element, enclosed in square brackets (`[]`). When you create an array using an array literal, it is initialized with the specified values as its elements, and its length is set to the number of arguments specified.

The following example creates the `coffees` array with three elements and a length of three:

```
var coffees = ["French Roast", "Colombian", "Kona"];
```

Note : An array literal is a type of object initializer. See Using Object Initializers.

If an array is created using a literal in a top-level script, JavaScript interprets the array each time it evaluates the expression containing the array literal. In addition, a literal used in a function is created each time the function is called.

Array literals are also `Array` objects. See `Array` and Indexed collections for details on `Array` objects.

Extra commas in array literals

You do not have to specify all elements in an array literal. If you put two commas in a row, the array is created with `undefined` for the unspecified elements. The following example creates the `fish` array:

```
var fish = ["Lion", , "Angel"];
```

This array has two elements with values and one empty element (`fish[0]` is "Lion", `fish[1]` is `undefined`, and `fish[2]` is "Angel").

If you include a trailing comma at the end of the list of elements, the comma is ignored. In the following example, the length of the array is three. There is no `myList[3]`. All other commas in the list indicate a new element.

Note : Trailing commas can create errors in older browser versions and it is a best practice to remove them.

```
var myList = ['home', , 'school', ];
```

In the following example, the length of the array is four, and `myList[0]` and `myList[2]` are missing.

```
var myList = [ , 'home', , 'school'];
```

In the following example, the length of the array is four, and `myList[1]` and `myList[3]` are missing. Only the last comma is ignored.

```
var myList = ['home', , 'school', , ];
```

Understanding the behavior of extra commas is important to understanding JavaScript as a language, however when writing your own code: explicitly declaring the missing elements as `undefined` will increase your code's clarity and maintainability.

Boolean literals

The Boolean type has two literal values: `true` and `false`.

Do not confuse the primitive Boolean values `true` and `false` with the true and false values of the Boolean object. The Boolean object is a wrapper around the primitive Boolean data type. See `Boolean` for more information.

Integers

Integers can be expressed in decimal (base 10), hexadecimal (base 16), octal (base 8) and binary (base 2).

- Decimal integer literal consists of a sequence of digits without a leading 0 (zero).
- Leading 0 (zero) on an integer literal, or leading 0o (or 0O) indicates it is in octal. Octal integers can include only the digits 0-7.
- Leading 0x (or 0X) indicates hexadecimal. Hexadecimal integers can include digits (0-9) and the letters a-f and A-F.
- Leading 0b (or 0B) indicates binary. Binary integers can include digits only 0 and 1.

Some examples of integer literals are:

```
0, 117 and -345 (decimal, base 10)

015, 0001 and -0o77 (octal, base 8)

0x1123, 0x00111 and -0xF1A7 (hexadecimal, "hex" or base 16)

0b11, 0b0011 and -0b11 (binary, base 2)
```

For more information, see Numeric literals in the Lexical grammar reference.

Floating-point literals

A floating-point literal can have the following parts:

- A decimal integer which can be signed (preceded by "+" or "-"),

- A decimal point ("."),
- A fraction (another decimal number),
- An exponent.

The exponent part is an "e" or "E" followed by an integer, which can be signed (preceded by "+" or "-"). A floating-point literal must have at least one digit and either a decimal point or "e" (or "E").

More succinctly, the syntax is:

```
[(+|-)][digits][.digits][(E|e)[(+|-)]digits]
```

For example:

```
3.1415926

-.123456789

-3.1E+12

.1e-23
```

Object literals

An object literal is a list of zero or more pairs of property names and associated values of an object, enclosed in curly braces (`{}`). You should not use an object literal at the beginning of a statement. This will lead to an error or not behave as you expect, because the { will be interpreted as the beginning of a block.

The following is an example of an object literal. The first element of the `car` object defines a property, `myCar`, and assigns to it a new string, "`Saturn`"; the second element, the `getCar` property, is immediately assigned the result of invoking the function `(carTypes("Honda"));` the third element, the `special` property, uses an existing variable (`sales`).

```
var sales = "Toyota";


function carTypes(name) {

  if (name === "Honda") {
    return name;

  } else {

    return "Sorry, we don't sell " + name + ".";

  }

}


var car = { myCar: "Saturn", getCar: carTypes("Honda"), special:
sales };


console.log(car.myCar);   // Saturn

console.log(car.getCar);  // Honda

console.log(car.special); // Toyota
```

Additionally, you can use a numeric or string literal for the name of a property or nest an object inside another. The following example uses these options.

```
var car = { manyCars: {a: "Saab", "b": "Jeep"}, 7: "Mazda" };


console.log(car.manyCars.b); // Jeep

console.log(car[7]); // Mazda
```

Object property names can be any string, including the empty string. If the property name would not be a valid JavaScript identifier or number, it must be enclosed in quotes. Property names that are not valid identifiers also cannot be accessed as a dot (`.`) property, but can be accessed and set with the array-like notation("`[]`").

```
var unusualPropertyNames = {

  "": "An empty string",

  "!": "Bang!"
}

console.log(unusualPropertyNames."");    // SyntaxError: Unexpected
string

console.log(unusualPropertyNames[""]);   // An empty string

console.log(unusualPropertyNames.!);     // SyntaxError: Unexpected
token !
console.log(unusualPropertyNames["!"]); // Bang!
```

In ES2015, object literals are extended to support setting the prototype at construction, shorthand for `foo: foo` assignments, defining methods, making super calls, and computing property names with expressions. Together, these also bring object literals and class declarations closer together, and let object-based design benefit from some of the same conveniences.

```
var obj = {

    // __proto__

    __proto__: theProtoObj,

    // Shorthand for ‘handler: handler’

    handler,

    // Methods

    toString() {

     // Super calls

     return "d " + super.toString();

    },

    // Computed (dynamic) property names

    [ 'prop_' + (() => 42)() ]: 42
};
```

Please note:

```
var foo = {a: "alpha", 2: "two"};

console.log(foo.a);     // alpha
console.log(foo[2]);   // two
```

```
//console.log(foo.2);  // Error: missing ) after argument list
```

```
//console.log(foo[a]); // Error: a is not defined
```

```
console.log(foo["a"]); // alpha
```

```
console.log(foo["2"]); // two
```

RegExp literals

A regex literal is a pattern enclosed between slashes. The following is an example of an regex literal.

```
var re = /ab+c/;
```

String literals

A string literal is zero or more characters enclosed in double (`"`) or single (`'`) quotation marks. A string must be delimited by quotation marks of the same type; that is, either both single quotation marks or both double quotation marks. The following are examples of string literals:

```
"foo"
```

```
'bar'
```

```
"1234"
```

```
"one line \n another line"
```

```
"John's cat"
```

You can call any of the methods of the String object on a string literal value—JavaScript automatically converts the string literal to a temporary String object, calls the method, then discards the temporary String object. You can also use the `String.length` property with a string literal:

```
console.log("John's cat".length)
```

```
// Will print the number of symbols in the string including
whitespace.

// In this case, 10.
```

In ES2015, template literals are also available. Template strings provide syntactic sugar for constructing strings. This is similar to string interpolation features in Perl, Python and more. Optionally, a tag can be added to allow the string construction to be customized, avoiding injection attacks or constructing higher level data structures from string contents.

```
// Basic literal string creation

`In JavaScript '\n' is a line-feed.`


// Multiline strings

`In JavaScript this is

 not legal.`


// String interpolation

var name = "Bob", time = "today";

`Hello ${name}, how are you ${time}?`


// Construct an HTTP request prefix is used to interpret the
replacements and construction

POST`http://foo.org/bar?a=${a}&b=${b}

     Content-Type: application/json

     X-Credentials: ${credentials}

     { "foo": ${foo},
       "bar": ${bar}}`(myOnReadyStateChangeHandler);
```

You should use string literals unless you specifically need to use a String object. See `String` for details on `String` objects.

Using special characters in strings

In addition to ordinary characters, you can also include special characters in strings, as shown in the following example.

```
"one line \n another line"
```

The following table lists the special characters that you can use in JavaScript strings.

Character	**Meaning**
`\0`	Null Byte
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Tab

`\v`	Vertical tab
`\'`	Apostrophe or single quote
`\"`	Double quote
`\\`	Backslash character
`\`*XXX*	The character with the Latin-1 encoding specified by up to three octal digits *XXX* between 0 and 377. For example, \251 is the octal sequence for the copyright symbol.
`\x`*XX*	The character with the Latin-1 encoding specified by the two hexadecimal digits *XX* between 00 and FF. For example, \xA9 is the hexadecimal sequence for the copyright symbol.

`\u`*`XXXX`*	The Unicode character specified by the four hexadecimal digits *XXXX*. For example, \u00A9 is the Unicode sequence for the copyright symbol. See Unicode escape sequences.
`\u{`*`XXXXX`*`}`	Unicode code point escapes. For example, \u{2F804} is the same as the simple Unicode escapes \uD87E\uDC04.