Common JavaScript Data Types

- Objects (which include arrays and functions).
- Numbers (no integers). Arithmetic based on 64-bit IEEE-754 standard.
- Strings.
- `undefined` and `null`.
- Booleans: `true` and `false`.

Objects are non-primitive. All other types are primitive types.

Numbers

No integers in early JS; problematic for financial calculations.

```
$ nodejs
> 1/0
Infinity
> 0/0
NaN
> NaN === NaN //IEEE behavior; in other languages too
false
> 2**53  //** is exponentiation operator
9007199254740992
> 2**53 + 1
9007199254740992  //IEEE 64 bit floats have a 53-bit mantissa.
> (2**53 + 1) === 2**53
true
> BigInt(2**53) + 1n //exact big integers
9007199254740993n
```

Normal Arithmetic and Bitwise Operators

- Usual *arithmetic operators* + (both infix and prefix), - (both infix and prefix), *, / and % (remainder, **has sign of dividend**), ** (power).
- *Bitwise operators* &, |, ^, ~, <<, >> (arith), >>> (logical).
- Bitwise operators always work with 32-bit 2's-complement integers.
- Previous property used in asm.js to obtain access to more efficient machine integer operations.

Arithmetic Operators Examples

```
> 123_456*2          //'_' allowed for readability
246912
> -77%13             //% has sign of dividend
-12
> 77%-13
12
> 2**2**3            //** is right associative
256
> (2**2)**3
64
> 18*2 + 77%13 / 2 //other binary operators left associative
42
```

Bitwise Operators Examples

```
> 1 | 2          //bitwise-or
3
> 0x99 & 0x3 //bitwise-and; hex notation
1
> 5 ^ 7          //bitwise-xor
2
> ~0              //bitwise-complement
-1              //0xffffffff is -1
> 3 << 4         //left-shift
48              //x << n === x * 2**n
> 100 >> 3      //arithmetic right-shift
12              // x>> n === x / 2**n
>
```

More on Shift Operators

- Shift operators can be used to multiply (left-shift) and divide (right-shift) by powers-of-2.
- Distinguish between >> (sign-propagating or arithmetic right-shift) and >>> (zero-fill or logical right-shift). No difference for non-negative numbers, but different results for negative numbers:

```
> -9 >> 1
-5
> -9 >>> 1
2147483643
> (-9 >>> 1).toString(16)
'7ffffffb'
>
```

Strings

- Strings are immutable.
- Classically, string literals are delimited using either double quotes " or single quotes '. Prefer ' delimiters since easier to type on normal keyboards. Backslashes interpreted as usual. Cannot span multiple lines.

```
> 'a' + 'b'      //string concatenation
'ab'
> 'abc'[1]       //indexing: results in string of length 1
'b'
> 'hello world'.indexOf('o')
4
> 'hello world'.lastIndexOf('o')
7
> 'hello world'.substr(3, 4)   //args: (startIndex, length)
'lo w'                         //treat as legacy function
```

Strings Continued

```
> 'hello world'.substring(3, 4) //args:(startIndex, endIndex)
'l'
> 'hello world'.slice(6)
'world'
> 'hello world'.slice(1, 4) //args: (startIndex, endIndex)
'ell'
> 'hello world'.slice(-3) //index from right; -1 is rightmost
'rld'
> 'hello world'.slice(-3, -1)
'rl'
```

Template String Literals

Enclosed within back-quotes ‘. Relatively new addition. Can contain direct newlines. All popular scripting languages have similar concepts (though introduced relatively recently to Python).

```
> const x = 22
undefined
> ‘The answer is ${x + 20}‘
’The answer is 42’
> ‘Betty bought a bit of butter
... ‘
’Betty bought a bit of butter\n’
> ‘Twas brillig and the slithy toves
... Did gyre and gimble in the wabe:‘
’Twas brillig and the slithy toves\nDid gyre and
gimble in the wabe:’
>
```

undefined

`undefined` Means lack of a value.

- Uninitialized variables are `undefined`.
- Missing parameters are `undefined`.
- Non-existent properties are `undefined`.
- Functions return `undefined` if no explicit return value.
- Use `x === undefined` to check if `x` is undefined.

undefined Continued

```
> let x                  //statement
undefined                //statement has no value
> x                       //expression
undefined                //value of expression
> x = {}                  //assignment expr; empty object
{}
> x.a
undefined
> undefined
undefined
> undefined = 1  //not a reserved word
1
> undefined               //immutable in global scope
undefined
```

`null`

`null` is a special value used to denote *no object*.
Can be used wherever an object is expected to indicate absence of an object. Examples:

- Parameters.
- Last object in a object chain.
- Use `x === null` to check if `x` is null.

Problems with null

Programmers are really sloppy with **null**:

- Tony Hoare has called **null** his "billion dollar mistake".
- Oracle does not distingush between **null** and an empty string!
- *Wired story* on living with a last name **Null**.
- NULL license plate problems.

typeof

Operator `typeof` used for categorizing primitives:

```
> typeof null
'object'
> typeof undefined
'undefined'
> typeof ""
'string'
> typeof 1
'number'
> typeof 1.2
'number'
> typeof true
'boolean'
```

typeof Continued

```
> typeof {}   //empty object literal
'object'
> typeof []   //empty array literal
'object'
> typeof (new Date())
'object'
>
```

instanceof

The `typeof` operator does not distinguish between different object types. Use `instanceof` operator for categorizing objects. The expression *v* `instanceof` *Type* returns true iff the constructor function Type was used to create *v*.

```
> ({} instanceof Object)
true
> [] instanceof Array
true
> [] instanceof Object
true
> (new Date()) instanceof Date
true
> (new Date()) instanceof Array
false
> (new Date()) instanceof Object
true
```

What is Truth

Many languages, particularly scripting languages, treat some set of values as *false* and **all other values** as *true*.
The *falsy* values in js are the following:

1. `undefined`.
2. `null`.
3. `false`.
4. 0.
5. "" (empty string).
6. `NaN` (Not-a-Number).

All other values are *truthy* and considered equivalent to true when used in a boolean context.

Comparison Operators

- Equality checking operators ==, !=, ===, !==. **Only use the last two**.
- >, <, >=, <= can be used with both numbers and strings.
- Objects compared by identity.

```
> 12.2 < 12.1
false
> 1 == true       //surprise: DO NOT USE!!
true
> 1 === true      //less surprising
false
> 'abc' < 'ab'
false
> 'abc' < 'abcd'
true
> {} === {}
false
```

Logical Operators

- Logical operators ! returns a strict boolean value (true or false).
- short-circuit && and short-circuit || return falsy/truthy values (last value evaluated).

```
> !true
false
> !1
false
> !!1   //common idiom used to convert to proper boolean
true
> !!0
false
```

Logical Operators Continued

```
> 'hello' || 'world'
'hello'
> 'hello' && 'world'
'world'
```

Default Values

Common idiom for default initialization:

```
> let x
undefined
> let y = x || 42
undefined
> y
42
```

But problematic because reasonable values like `0`, `''` and `false` are falsy values:

```
> x = 0
undefined
> y = x || 42
42   //y assigned 42 even tho' x has a reasonable value
```

Logical Operators Continued

Default initialization idiom should only be used if a valid value is not one of the falsy values.

```
> x = 0   //0 is falsy
0
> let z = x || defaultValue
undefined
> z
42  //z assigned defaultValue despite x having a value
```

Nullish Coalescing Operator

nullish coalescing operator ?? returns right operand when left operand is **nullish**, i.e. **null** or **undefined**, otherwise it returns its left operand (relatively new addition to JavaScript):

```
> x = 0
0
> y = x ?? 42
0
> x = undefined
undefined
> y = x ?? 42
42
```

Modern way to do default initialization.

Optional Object Chaining

When chaining accesses, we often need to check that intermediate values are not **nullish** (i.e. **null** or **undefined**).

```
const c = obj && obj.a && obj.a.b && obj.a.b.c;
```

Can be done more compactly using new feature *optional chaining* `.?` operator:

```
const c = obj?.a?.b?.c;   //undefined if any accessor nullish
```

Syntax also allows:

```
a?.[expr]          //dynamic property name
f?.(arg1, arg2)    //returns undefined if f nullish
```

Control Constructs

- *Condition-based selection* using **if** and **if-else** statements. No surprises except truthy interpretation of condition.
- *Multiway selection* on a value (including string values) using **switch-case-default**. Value compared with **case**-values using ===. Control will fall-through from one **case** to the next, unless there is a **break** statement.
- Looping using while. Body may not execute at all if condition is initially falsy.
- Looping using do-while statement executes its body at least once, irrespective of the value of the condition.

For Loops

- *Traditional for* loop with initialization expression, condition expression and increment expression. Any of the three expressions can be omitted.
- Looping through *object properties* using **for-in**.
- Looping over *iterable objects* like arrays using **for-of**.

For Loop Examples

Summing positive elements of array a (better to use `filter` and `reduce`):
Using traditional **for**:

```
let sum = 0;
for (let i = 0; i < a.length; i++) {
  if (a[i] > 0) sum += a[i];
}
```

Using **for-of**:

```
let sum = 0;
for (const v of a) {
  if (v > 0) sum += v;
}
```

Loop Choice

Always use loop which moves as much of loop control into loop header; do so at the highest level of abstraction. In descending order of preference:

- Looping through array: use **for-of**. Looping through object properties: use **for-in**.
- Looping through integer range: use traditional **for**.
- Body executed at least once: use **do-while**.
- Plain **while** loop is most general; lowest preference since loop update hidden within loop body.

Functions

- Functions are **first-class**: need not have a name ("anonymous"), can be passed as parameters, returned as results, stored in data structure.
- Functions can be nested within one another.
- **Closures** preserve the referencing environment of a function.
- During execution of a function, there is always an implicit object, referred to using `this`. The word `this` will be pronounced `self` when speaking.

Function Definitions

- Traditional function definitions

```
function max1(a, b) { return a > b ? a : b }
```

- Anonymous function using **function** keyword

```
max2 = function(a, b) { return a > b ? a : b }
```

- Anonymous fat-arrow function

```
> x = max4 = (a, b) => a > b ? a : b
> (a => { const v = max4(a, 42);
          return v * 2;
        }
  )(5)   //IIFE
84
```

- Subtle differences in semantics.

Arrays

Arrays (AKA lists) are like objects except:

- It has an auto-maintained `length` property (always set to 1 greater than the largest array index).
- Arrays have their prototype set to `Array.prototype` (`Array.prototype` has its prototype set to `Object.prototype`, hence arrays inherit object methods).

Array Examples

```
> a = []
[]
> a[999] = 22
22
> a
[ <999 empty items>, 22 ]
> a[999]
22
> a.length = 1    //truncates
1
> a[999]
undefined
```

Array Examples Continued

```
> a[2] = 22
22
> a.length
3
> a.join('|')
'||22'
> a.x = 99   //arrays are objects: can have properties
99
> a
[ <2 empty items>, 22, x: 99 ]
> a.constructor
[Function: Array]
>
```

Spreading Arrays

Arrays can be spread into array literals or function calls using the ... **spread** operator:

```
> a = [3, 4, 5]
[ 3, 4, 5 ]
> [33, 44, ...a, 66]
[ 33, 44, 3, 4, 5, 66 ]
> ((a, b) => a * b)(a)
NaN
> ((a, b) => a * b)(...a)
12
> ((a, b) => a * b)(5, ...a)
15
```

Mapping Arrays

The `map()` function returns a new array which is the result of calling its argument function on each element of the calling array.

```
> function times3(x) { return 3*x; }
undefined
> [1, 2, 3].map(times3)
[ 3, 6, 9 ]
> [1, 2, 3].map(x => 7*x);
[ 7, 14, 21 ]
> [7, 3, 2, 4].map(x => x % 2 === 0)
[ false, false, true, true ]
>
```

Reducing Arrays

The `reduce()` function using a function `f(accumulator, element)` to reduce an array to a single value.

```
> [1,2,3,4,5].reduce((acc, value) => acc + value)
15
> [1,2,3,4,5].reduce ((acc, value) => acc + value, 7 )
22
> [12].reduce((acc, value) => acc + value)
12
> > [].reduce((acc, value) => acc + value, 15)
15
> [].reduce((acc, value) => acc + value)
TypeError: Reduce of empty array with no initial value
...
```

Applying a Function to Each Array Element

`forEach()` applies function to each element. Like many other functions callback takes 3 arguments: elementValue, elementIndex plus full array.

```
indexes = []
[]
> [1, 2, 3, 4].forEach(( v, i ) => {
    if (v%2 === 0) indexes.push (i);
   })
undefined
> indexes
[ 1, 3 ]
>
```

Other Higher-Order Array Functions

Includes `every()`, `find()`, `findIndex()`, `filter()`, `reduceRight()`, `some()`.

```
> [1, 2, 3, 4].find(x => x%2 === 0)
2
> [1, 2, 3, 4].findIndex(x => x%2 === 0)
1
> [1, 2, 3, 4].every(x => x%2 === 0)
false
> [1, 2, 3, 4].some(x => x%2 === 0)
true
> [1, 2, 3, 4].reduce((acc, v) => acc - v)
-8                  //(((1-2)-3)-4
> [1, 2, 3, 4].reduceRight((acc, v) => acc - v)
-2                  //1-(2-(3-4))
>
```

Other Higher-Order Array Functions

Summing positive elements of array:

```
> [1, -2, 3, -4].filter((e) => e > 0).
    reduce((acc, e) => acc + e, 0)
4
```

Function Arguments

- No requirement that number of actual arguments agree with the number of declared formal parameters.
- If the number of actual arguments is greater than the number of formal parameters, then the extra arguments are ignored:

```
> ((a, b) => a + b)(3, 4, 5)
7
```

- If the number of actual arguments is less than the number of formal parameters, then the extra formal parameters are **undefined**. Dangerous!!

```
> ((a, b) => a > b)(3)
false
```

- The `length` of a function is its number of declared formal parameters:

```
> ((a, b) => a > b).length
2
```

Accessing Actual Arguments

- The actual arguments to a function defined using the **function** keyword are accessible using a pseudo variable `arguments`:

```
> (function() { return arguments.length ; })(3,
7)
2
```

- The `arguments` pseudo-variable is **array-like**; it supports `length` and indexing operations but is not a real array.

```
> (function() { return -arguments[0] ; })(3, 7)
-3
> ( function() {
      return arguments.map(x => x + 1);
    }
  )(3, 7)
Uncaught TypeError: arguments.map is not a
                    function
```

Accessing Actual Arguments Continued

- `arguments` can be converted to a real array by being **spread** into a real array:

```
> ( function() {
      return [...arguments].map(x => x + 1);
    }
  )(3, 7)
[ 4, 8 ]
```

- Modern JS needs to use `arguments` less; instead, if the last argument is preceeded by `...`, then all **rest** arguments are collected into that parameter as a **real array**. Works with fat-arrow functions too!

```
> ((a, ...args) => args.map(x => x*2))
  (3, 4, 5, 6)
[ 8, 10, 12 ]
```