JavaScript CheatSheet

JavaScript is what everyone calls the language, but that name is trademarked (by Oracle, which inherited the trademark from Sun). Therefore, the official name of JavaScript is ECMAScript. The "ECMA" in "ECMAScript" comes from the organisation that hosts the primary standard, the European Computer Manufacturers Association.

As the programming language of browsers, it is remarkably error-tolerant. It simply "fails silently" by giving error values such as `undefined` when things are not there or `0 / 0` $\approx$ `NaN` for nonsensical numeric expressions.

By accident, there are two (mostly) *interchangeable* values `null` and `undefined` that denote the absence of a meaningful value. Many operations that don't produce meaningful values yield `undefined` simply because they have to yield *some* value. Here is a neat story about null.

Types

JavaScript considers types only when actually running the program, and even there often tries to implicitly convert values to the type it expects.

- `typeof` gives a string value naming the type of its argument.
- The functions `Number, String, Boolean` try to convert values into those types.

```
console.log(typeof 4.5, typeof '4.5', typeof true)
// ⇒ number string boolean

console.log(8 * null // Multiplication needs numbers so null ↦ 0
            , 'five' * 2 // 'five' is not a number, so 'five' ↦ NaN
            , '5' - 1    // Subtraction needs numbers so '5' ↦ 5
            , '5' + 1)   // The first is a string,
                         // so "+" denotes catenation, so 1 ↦ '1'

console.log(Number('2.3')   // ⇒ 2.3
           ,Number('five')  // ⇒ NaN
           ,Boolean('five') // ⇒ true
           ,Boolean('')     // ⇒ false
           ,String(NaN)     // ⇒ 'NaN'
           ,String(null))   // ⇒ 'null'
```

```
2.3 NaN true false NaN null
```

Variable Bindings

`let` $\mathtt{x}_0$ `=` $\mathtt{v}_0$`, ...,` $\mathtt{x}_n$ `=` $\mathtt{v}_n$`;` introduces n-new names $\mathtt{x}_i$ each having value $\mathtt{v}_i$.

- The $\mathtt{v}_i$ are optional, defaulting to `undefined`.
- The program crashes if any $\mathtt{x}_i$ is already declared.
- Later we use $\mathtt{x}_i$ `=` $\mathtt{w}_i$`;` to update the name $\mathtt{x}_i$ to refer to a new value $\mathtt{w}_i$.
  - Augmented updates: `x` $\oplus$`= y` $\equiv$ `x = x` $\oplus$ `y`
  - Increment: `x--` $\equiv$ `x += 1`
  - Decrement: `y--` $\equiv$ `x -= 1`

```
let x, y = 1, z;
console.log(x, y, z); // ⇒ undefined 1 undefined
```

- In the same way, for the same purpose, we may use `var` but it has undesirable properties; e.g., its declarations are in the global scope and no error is raised using `var x = ⋯` if `x` is already declared.
- In the same way, we may use `const` to introduce names that are constant: Any attempt to change their values crashes the program.
- A binding name may include dollar signs ($) or underscores (_) but no other punctuation or special characters.

Scope and Statements

Each binding has a scope, which is the part of the program in which the binding is visible. For bindings defined outside of any function or block, the scope is the whole program—you can refer to such bindings wherever you want. These are called global.

```
let x = 10;

{ // new local scope
  let y = 20;
  var z = 30;
  console.log(x + y + z); // ⇒ 60
}

// y is not visible here
// console.log(y)
```

```
// But z is!
console.log(x + z); // ⇒ 40
```

⊙ *global bindings* are defined outside of any block and can be referenced anywhere.

⊙ *local bindings* are defined within a block and can only be referenced in it.

⊙ `let, const` declare local bindings; `var` always makes global ones!

Besides the assignment statement, we also have the following statements:

- ◇ Conditionals: `if (condition) A else B`
- ◇ Blocks: If S_i are statements, then `{`S_0`; ...; `S_n`;}` is a statement.
- ◇ The `for/of` syntax applies to arrays, strings, and other iterable structures —we will define our own later.

```
// Print all the elements in the given list.
for (let x of ['a', 1, 2.3]) {
  console.log(`x ≈ ${x}`);
  }
```

JavaScript is whitespace insensitive.

Arithmetic

In addition to the standard arithmetic operations, we have `Math.max(`x_0`, ..., `x_n`)` that takes any number of numbers and gives the largest; likewise `Math.min(···)`. Other common functions include `Math.sqrt, Math.ceil, Math.round, Math.abs,` and `Math.random()` which returns a random number between 0 and 1. Also, use `%` for remainder after division.

```
// Scientific notation: xey ≈ x × 10^y
console.log(1, 2.998e8, 100 + 4 * 11)

// Special numbers so that division "never crashes".
console.log(1/0, -1/0, Infinity - 10) // ⇒ Infinity -Infinity Infinity
console.log(Infinity - Infinity, 0/0) // ⇒ NaN NaN

// Random number in range min...Max
Math.floor(Math.random() * (max - min) + min)
```

`NaN` stands for "not a number", it is what you get when a numeric expression has no meaningful value.

- ◇ Any `NaN` in an arithmetic expressions swallows the whole expression into a `NaN`.
- ◇ `Number.isNaN(x)` is true iff `x` is `NaN`.

Everything is equal to itself, except `NaN`. Why? `NaN` denotes the result of nonsensical computations, and so is not equal to the result of any other nonsensical computation.

```
console.log(NaN == NaN) // ⇒ false
```

```
false
```

Booleans

The empty string `''`, list `[]`, and `0, NaN` are falsey —all else is truthy.

- ◇ Note: `(p < q < r)` ≈ `(p < q) < r`, it is not conjunctive!

```
console.log(true, false, 3 > 2, 1 < 2, 1 != 2, 4 <= 2 < 3)

// Upper case letters come first, then lower case ones.
console.log('abc' < 'def', 'Z' < 'a')

// Equality with coercions, and without.
console.log(1.23 == '1.23', 1.23 === '1.23')
```

```
true false true true true true false
true true
true false
```

- ◇ *Precise Equality* `===` is equality with no type coercions.
- ◇ Applying the "not" `!` operator will convert a value to Boolean type before negating it.
- ◇ Precedence: Relationals like `==` and `>` are first, then "and" `&&`, then "or" `||`.
- ◇ The ternary operator: `condition ? if_true : if_false`

```
console.log(null == undefined) // ⇒ true
```

```
true
```

Only the empty values are coerced into being equal, no other value is equal to an empty value. As such, `x != null` means that `x` is not an empty value, and is in fact a real meaningful value.

Since `&&` and `||` are lazy, `x || y` means return `x` if `x != false` and otherwise return `y`; i.e., *give me x if it's non-empty, else y.*

Likewise, `x && y` means *give me y, if x is nonempty, else give me the particular empty value x.*

```
console.log( 4  == 3   && 4   // 3 is truthy
           ,''  == ''  && 4   // '' is falsey
           ,'H' == 'H' && 4   // 'H' is truthy
           , 0  == 0   && 4   // 0 is falsey
           , 4  == 0   || 4   // 0 is falsey
           )
```

Strings

Any pair of matching single-quotes, backticks, or double-quotes will produce a string literal. However, backticks come with extra support: They can span multiple lines and produce *formatted strings*, where an expression can be evaluated if it is enclosed in `${···}`.

```
console.log(`half of 100 is ${100 / 2}`) // ⇒ half of 100 is 50
```

- `s.repeat(n)` ≈ Get a new string by gluing n-copies of the string s.
- Trim removes spaces, newlines, tabs, and other whitespace from the start and end of a string.

  ```
  console.log("  okay \n ".trim()); // ⇒ okay
  ```
- `s.toUpperCase()` and `s.toLowerCase()` to change case.
- `s.padStart(l, p)` ≈ Ensure s is of length $\geq l$ by padding it with p at the start.

  ```
  console.log(String(6).padStart(3, "0")); // ⇒ 006
  ```
- `s.replace(/./g, c => p(c) ? f(c) : '')` ≈ Keep only the characters that satisfy predicate `p`, then transform them via `f`.

  ```
  let s = 'abcde'.replace(/./g, c => 'ace'.includes(c) ? c.toUpperCase() : '')
  console.log(s); // ⇒ ACE
  ```

The following methods also apply to arrays.

- `s.length` ⇒ Length of string
- `s[i]` ⇒ Get the i-th character from the start
  - Unless `0` $\leq i <$ `s.length`, we have `s[i] = undefined`.
- `s.concat(t)` ⇒ Glue together two strings into one longer string; i.e., `s + t`.

  ```
  console.log(('cat' + 'enation').toUpperCase()) // ⇒ CATENATION
  ```
- `s.includes(t)` ⇒ Does `s` contain `t` as a substring?
- `s.indexOf(t)` ⇒ Where does substring `t` start in `s`, or -1 if it's not in `s`.
  - To search from the end instead of the start, use `lastIndexOf`.
- `s.slice(m,n)` ⇒ Get the substring between indices m (inclusive) and n (exclusive).
  - n is optional, defaulting to `s.length`.
  - If n is negative, it means start from the end: `s.slice(-`n`)` ≈ `s.slice(s.length -` n`)`.
  - `s.slice()` ⇒ Gives a copy of `s`.

- There is no character type, instead characters are just strings of length 1.
- You can "split" a string on every occurrence of another string to get a list of words, and which you can "join" to get a new sentence. `s.split(`d`).join(`d`)` ≈ s.
- To treat a string as an array of characters, so we can apply array only methods such as `f = reverse`, we can use split and join:

  `s.split('').f().join('')`
- Keeping certain characters is best done with regular expressions.

Arrays

Array indexing, `arr[i]`, yields the i-th element from the start; i.e., the number of items to skip; whence `arr[0]` is the first element.

```
let numbers = [];

numbers.push(2);
numbers.push(5);
numbers.push(7);

// or
numbers = [2, 5, 7]

console.log(numbers[2]); // ⇒ 7
let last = numbers.pop() // ⇒ 7
console.log(numbers) // ⇒ [ 2, 5 ]

// ⇒ undefined
console.log(numbers[2]);
```

```
// Is an element in the array? No.
console.log(numbers.includes(7))

numbers  = numbers.concat('ola')
console.log(numbers)
// ⇒ [ 2, 5, 'ola' ]

console.log(numbers.reverse())
// ⇒ [ 'ola', 5, 2 ]
7
[ 2, 5 ]
undefined
false
[ 2, 5, 'ola' ]
[ 'ola', 5, 2 ]
```

(**Stack**) The push method adds values to the end of an array, and the pop method does the opposite, removing the last value in the array and returning it. (**Queue**) The corresponding methods for adding and removing things at the start of an array are called `unshift` and `shift`, respectively.

Arrays have the following methods, which behave similar to the string ones from earlier.

`length` `concat` `includes` `indexOf` `lastIndexOf` `slice`

One difference is that unlike string's `indexOf`, which searches for substrings, array's `indexOf` searches for a specific value, a single element.

```
console.log([1, 2, 3, 2, 1].indexOf(2));     // ⇒ 1
console.log([1, 2, 3, 2, 1].lastIndexOf(2)); // ⇒ 3
```

The `concat` method can be used to glue arrays together to create a new array, similar to what the `+` operator does for strings.

- ◇ If you pass `concat` an argument that is not an array, that value will be added to the new array as if it were a one-element array. This is a `push`.

`Array(`n`).fill(`x`)` ≈ Get a new array of n-copies of element x.

- ◇ `xs.forEach(a)` to loop over the elements in an array and perform action `a`.
- ◇ `xs.filter(p)` returns a new array containing only the elements that pass the predicate `p`.
- ◇ `xs.map(f)` transforms an array by putting each element through the function `f`.
- ◇ `xs.reduce(f, e)` combines all the elements in an array into a single value.
  - ○ We can omit the starting value `e` if the array `xs` is non-empty, in which case `e` is taken to be the first element `xs[0]`.
- ◇ `xs.some(p)` tests whether any element matches a given predicate function `p`.
  - ○ `xs.every(p)` tests if every element of `xs` satisfies `p`.
- ◇ `xs.findIndex(p)` finds the position of the first element that matches the predicate `p`.

With the exception of `forEach`, the above functions do not modify the array they are given.

```
// Print the elements of the given array
['a', 'b', 'c'].forEach(l => console.log(l));

// ∃/∀
console.log([1, 2, 3].some(e => e == 2))  // true
console.log([1, 2, 3].every(e => e == 2)) // false

// Sum the elements of an array
console.log([1, 2, 3, 4].reduce((soFar, current) => soFar + current)); // ⇒ 10

// flatten an array of arrays
let flatten = (xss) => xss.reduce((sofar, xs) => sofar.concat(xs), [])

let arrays = [[1, 2, 3], [4, 5], [6]];
console.log(flatten(arrays)) // ⇒ [ 1, 2, 3, 4, 5, 6 ]
```

Higher-order functions start to shine when you need to compose operations.

Functions

Function values can do all the things that other values can do; i.e., they can be used in arbitrary expressions; e.g., a binding that holds a function is still just a regular binding and can, if not constant, be assigned a new value.

- ◇ A `function` definition is a regular binding where the value of the binding is a function.

  Functions declared using the top-level `function` keyword may be used before their declarations.

```
const square = function(x) {
  return x * x;
};

console.log(square(12)); // ⇒ 144
```

```
// Shorter way to define functions
console.log(square2(12));
function square2(x) {
  return x * x;
}
```

- A `return` keyword without an expression after it will cause the function to return `undefined`.
- Functions that don't have a `return` statement at all, similarly return `undefined`.
- Declaring `function f (···) {···}` will not raise a warning if the name `f` is already in use —similar to `var`.
- One may also define functions using "arrow" notation: `(x0, ..., xn) => ···`.
  - When there is only one parameter name, you can omit the parentheses around the parameter list.
  - If the body is a single expression, rather than a (multi-line) block in braces, that expression will be returned from the function.

  So, these two definitions of square do the same thing:

  ```
  const square1 = (x) => { return x * x; };
  const square2 =  x  => x * x;
  ```

As will be seen, arrow functions are not exactly the same as declared functions.

JavaScript is extremely fault-tolerant: If we give a function more arguments than it needs, the extra arguments are just ignored. If we give it too few arguments, the missing arguments are assigned `undefined`.

```
// Extra arguments are ignored
console.log(square(4, true, "hedgehog")); // ⇒ 16

// No longer a function!
square = 'g'
```

(**Default Values**) If you write an = operator after a parameter, followed by an expression, the value of that expression will replace the argument when it is not given.

```
let square = (x = 1) => x * x;
console.log(square(3)); // ⇒ 9
console.log(square());  // ⇒ 1
```

(**Rest Parameters**) It can be useful for a function to accept any number of arguments. For example, `Math.max` computes the maximum of all the arguments it is given. To write such a function, you put three dots before the function's last parameter, which is called "the rest parameter" and it is treated as an array containing all further arguments.

```
function max(...numbers) {
  let result = -Infinity;
  for (let number of numbers) {
    if (number > result)
      result = number;
  }
  return result;
}
```

```
console.log(max(4, 1, 9, -2));  // ⇒ 9
```

You can use a similar three-dot notation to call a function with an array of arguments.

```
let numbers = [5, 1, 7];
console.log(max(...numbers));
// ⇒ 7
```

This "spreads" out the array into the function call, passing its elements as separate arguments. It is possible to include an array like that along with other arguments, as in `max(9, ...numbers, 2)`.

Higher-order functions allow us to abstract over actions, not just values. They come in several forms.

For example, we can have functions that create new functions.

```
let greaterThan   = n => (m => m > n);
let greaterThan10 = greaterThan(10);
console.log(greaterThan10(11)); // ⇒ true
```

And we can have functions that change other functions. (**Decorators**)

```
function noisy(f) {
  return (...args) => {
    let result = f(...args);
    console.log(`Called: ${f.name}(${args}) ≈ ${result}`);
    return result;
  };
}
noisy(Math.min)(3, 2, 1); // Called: min(3,2,1) ≈ 1
```

We can even write functions that provide new types of control flow.

```
function unless(test, then) {
  if (!test) then();
}
let n = 8;
```

```
unless(n % 2 == 1, () => {
    console.log(n, "is even");
  });
// ⇒  8 is even
```

Destructuring and the "spread" Operator

If you know the value you are binding is an array/object, you can use []/{} brackets to "look inside" of the value, binding its contents.

One of the reasons the `doit` function below is awkward to read is that we have a binding pointing at our array, but we'd much prefer to have bindings for the elements of the array, whence the second definition of `doit`.

```
let xs = [9, 11, 22, 666, 999];

// The following are the same.
function doit(xs){ return xs[0] + xs[1] + xs[2]; }
function doit([x, y, z]) {return x + y + z; }
//
// Only first three items accessed in "doit"; extra args are ignored as usual.
console.log(doit(xs))

// Destructuring to get first three elements and remaining
let x = xs[0], y = xs[1], z = xs[2], ws = xs.slice(3);
console.log(x, y, z, ws) // ⇒  9 11 22 [ 666, 999 ]
// Nice! Same thing.
let [a, b, c, ...ds] = xs
console.log(a, b, c, ds) // ⇒  9 11 22 [ 666, 999 ]

// Destructuring to get first and remaining elements
let [head, ...tail] = xs
console.log(head, tail) // ⇒  9 [ 11, 22, 666, 999 ]

// Destructuring on an object to get two properties and the remaining subobject
let {name, age, ...more} = {name: "Musa", age: 72, x: 1, y: 2}
console.log(name, age, more) // ⇒  Musa 72 { x: 1, y: 2 }

// Destructuring: Simultaneous assignment!
var p = 1, q = 2    // ⇒  1, 2
var [p, q] = [q, p] // swap them
console.log(p, q)   // ⇒  2, 1

// Unpacking: f(...[x0, ..., xn]) ≈ f(x0, ..., xn)
console.log(Math.min(...xs)) // ⇒  9

// Unpacking: Merging arrays/objects
let ys = [1, ...xs, 2, 3] // ⇒  1, 9, 11, 22, 666, 999, 2, 3
let zs = {w: 0, ...more, z: 3} // ⇒  { w: 0, x: 1, y: 2, z: 3 }

// Updating a property, a key-value pair
zs = {...zs, w: -1} // ⇒  { w: -1, x: 1, y: 2, z: 3 }
```

Note that if you try to destructure `null` or `undefined`, you get an error, much as you would if you directly try to access a property of those values.

```
   let {x0, ..., xn, ...w} = v
≡ let x0 = v.x0, ..., xn = v.xn; w = v; delete w.x0, ..., delete w.xn
```

As usual, in arrow functions, we may destructure according to the shape of the elements of the array; e.g., if they are lists of at least length 2 we use `(soFar, [x, y]) => ···`. This may be useful in higher order functions such as `map, filter, reduce`.

Objects

Objects and arrays (which are a specific kind of object) provide ways to group several values into a single value. Conceptually, this allows us to put a bunch of related things in a bag and run around with the bag, instead of wrapping our arms around all of the individual things and trying to hold on to them separately. These "things" are called *properties*.

Arrays are just a kind of object specialised for storing sequences of things.

Values of the type *object* are arbitrary collections of properties. One way to create an object is by using braces as an expression that lists properties as *"name:value"* pairs.

1. Almost all JavaScript *values* have properties. The exceptions are `null` and `undefined`. If you try to access a property on one of these nonvalues, you get an error. Properties are accessed using `value.prop` or `value["prop"]`.

2. Whereas `value.x` fetches the property of value named x, `value[e]` tries to evaluate the expression `e` and uses the result, converted to a string, as the property name.
3. The dot notation only works with properties whose names look like valid (variable) binding names. So if you want to access a property named `2` or `John Doe`, you must use square brackets: `value[2]` or `value["John Doe"]`.
4. Unless `value` contains a property `x`, we have `value.x` $\approx$ `undefined`.
   - ◇ Hence, out of bounds indexing results in `undefined`.
5. Notice that the `this` keyword allows us to refer to other parts of *this* object literal. Above, `info` used the `person` object's information, whereas `speak` did not. The "`this`" keyword is covered in more detail below.
6. Variables names in an object literal, like `languages`, denote a shorthand for a property with the same and value, but otherwise is no longer related to that binding.

   This is useful if we want multiple objects to have the same binding; e.g., with `let x = ···, a = {name: 'a', x}, b = {name: 'b', x}`, both objects have a `x` property: `a.x` and `b.x`.
7. We cannot dynamically attach new properties to the atomic types String, Number, Boolean; e.g., `let x = 2; x.vest = 'purple'; console.log(x.vest);` prints `undefined`. We can write it, but they "don't stick".
8. Below, we could have begun with the empty object then added properties dynamically: `let person = {}; person.name = 'musa'; person.age = 29; ....`

```
let languages = ['js', 'python', 'lisp']
let person = { name: 'musa'
             , age: 27
             , 'favourite number': 1
             , languages // Shorthand for "languages: ['js', 'python', 'lisp']"
             , age: 29  // Later bindings override earlier ones.
             // Two ways to attach methods; the second is a shorthand.
             , speak: () => `Salamun Alaykum! Hello!`
             , info () { return `${this.name} is ${this.age} years old!`; }
             };

console.log(person.age) // ⇒ 29

// Trying to access non-existent properties
// Reading a property that doesn't exist will give you the value undefined.
console.log(person.height) // ⇒ undefined

// Is the property "name" in object "person"?
console.log('name' in person); // ⇒ true

// Updating a (computed) property
let prop = 'favourite' + ' ' + 'number'
person['favourite number'] = 1792
console.log(person[prop]) // ⇒ 1792

// Dynamically adding a new property
person.vest = 'purple'
console.log(person.vest) // ⇒ purple

// Discard a property
delete person['favourite number']

// Get the list of property names that an object *currently* has.
console.log(Object.keys(person)) // ⇒ [ 'name', 'age', 'languages', 'vest' ]

// Variables can contribute to object definitions, but are otherwise unrelated.
languages = ['C#', 'Ruby', 'Prolog']
console.log(person.languages) // ⇒ [ 'js', 'python', 'lisp' ]

// Calling an object's methods
console.log(person.speak()) // ⇒ Salamun Alaykum! Hello!
console.log(person.info())  // ⇒ musa is 29 years old!
```

You can define getters and setters to secretly call methods every time an object's property is accessed. E.g., below `num` lets you read and write `value` as any number, but internally the getter method is called which only shows you the value's remainder after division by the modulus property.

```
let num = { modulus: 10
          , get value() { return this._secret % this.modulus; }
          , set value(val) { this._secret = val; } }

num.value = 99
console.log(num._secret) // ⇒ 99
console.log(num.value) // ⇒ 9
```

```
num.modulus = 12;
console.log(num.value) // ⇒ 3
```

◇ Exercise: Make an object `num` such that `num.value` varies, returning a random number less than 100, each time it's accessed.

Using `get, set` is a way to furnish prototypes with well-behaved properties that are readable or writable, or both.

An object can also be used as a *"key:value"* dictionary: When we 'look-up' a key, we find a particular value. E.g., with `ages = {mark: 12, james: 23, larry: 42}` we use `ages.mark` to find Mark's age.

Similarly, objects can be used to simulate *keyword arguments* in function calls.

The `this` Keyword

Usually a method needs to do something with the object it was called on. When a function is called as a method — looked up as a property and immediately called, as in `object.method()` —— the binding called `this` in its body automatically points at the object that it was called on.

```
function speak(line) {
  console.log(`The ${this.type} rabbit says '${line}'`);
}
let whiteRabbit  = {type: "white", speak};
let hungryRabbit = {type: "hungry", speak};

whiteRabbit.speak("Hola!"); // ⇒ The white rabbit says 'Hola!'
hungryRabbit.speak("Hey!")  // ⇒ The hungry rabbit says 'Hey!'
```

You can think of `this` as an extra parameter that is passed in a different way. If you want to pass it explicitly, you can use a function's `call` method, which takes the `this` value as its first argument and treats further arguments as normal parameters.

```
speak.call(hungryRabbit, "Burp!");
// ⇒ The hungry rabbit says 'Burp!'
```

With `call`, an object can use a method belonging to another object. E.g., below we use `whiteRabbit`'s speaking method with its `this` keywords referring to `exoticRabbit`.

```
let exoticRabbit = {type: 'exotic'}

whiteRabbit.speak.call(exoticRabbit, `Jambo!`)
// ⇒ The exotic rabbit says 'Jambo!'
```

Since each function has its own `this` binding, whose value depends on the way it is called, you cannot refer to the `this` of the wrapping scope in a regular function defined with the function keyword.

Arrow functions are different —they do not bind their own `this` but can see the `this` binding of the scope around them. Thus, you can do something like the following code, which references `this` from inside a local function:

```
function normalise() {
  console.log(this.coords.map(n => n / this.length));
}
normalise.call({coords: [0, 2, 3], length: 5}); // ⇒ [0, 0.4, 0.6]
```

If we had written the argument to `map` using the `function` keyword, the code wouldn't work.

Object-Oriented Programming

In English, *prototype* means a preliminary model of something from which other forms are developed or *copied*. As such, a *prototypical* object is an object denoting the original or typical form of something.

In addition to their properties, JavaScript objects also have prototype —i.e., another object that is used as a source of additional properties. When an object gets a request for a property that it does not have, its prototype will be searched for the property, then the prototype's prototype, and so on.

◇ `Object.getPrototypeOf(x)` returns the prototype of an object `x`.

For example, arrays are derived from `Array.prototype` which is derived from `Object.prototype` —which is the great ancestral prototype, the entity behind almost all object. `Object.prototype` provides a few methods that show up in all objects, such as `toString`, which converts an object to a string representation.

◇ We can use the `Object.getOwnPropertyNames(x)` to get all the property names linked to object `x`.

It is occasionally useful to know whether an object was derived from a specific class. For this, JavaScript provides a binary operator called `instanceof`. Almost every object is an instance of Object.

◇ x `instanceof` $y \approx$ `Object.getPrototypeOf(`x`) == `y`.prototype`

```
// "Object" includes "toString", and some other technical utilities.
console.log(Object.getOwnPropertyNames(Object.prototype))

// Some true facts
```

```
console.log( {}       instanceof Object
           , []       instanceof Array
           , Math.max instanceof Function
           , Math.max instanceof Object) // Since Function derives from Object

// "Object" has no parent prototype.
console.log(Object.getPrototypeOf(Object.prototype)); // ⇒ null
```

(**Extension Methods / Open Classes**) To attach a new property to a 'kind' of object, we simply need to attach it to the prototype —since all those 'kinds' of objects use the prototype's properties. Let's attach a new method that can be used with *any* array.

```
Array.prototype.max = function () {
   console.log('ola'); return Math.max(...this)
   }

console.log([3,1,5].max()); // ⇒ Prints "ola", returns 5

console.log(Object.getOwnPropertyNames(Array.prototype))
// ⇒ Includes length, slice, ..., and our new "max" from above
```

When you call the `String` function (which converts a value to a string) on an object, it will call the `toString` method on that object to try to create a meaningful string from it.

```
Array.prototype.toString = function() { return this.join(' and '); };
console.log(String([1, 2, 3])) // ⇒ 1 and 2 and 3
```

(**Overriding**) When you add a property to an object, whether it is present in the prototype or not, the property is added to the object itself. If there was already a property with the same name in the prototype, this property will no longer affect the object, as it is now hidden behind the object's own property.

```
Array.prototype.colour = 'purple'

let xs = [1, 2, 3]
console.log(xs.colour) // ⇒ purple
xs.colour = 'green'
```

```
console.log(xs.colour) // ⇒ green

console.log(Array.prototype.colour)
// ⇒ purple
```

You can use `Object.create` to create an object with a specific prototype. The default prototype is `Object.prototype`. For the most part, `Object.create(someObject)` ≈ `{ ...someObject }`; i.e., we *copy* the properties of `someObject` into an empty object, thereby treating `someObject` as a prototype from which we will build more sophisticated objects.

Unlike other object-oriented languages where `Object` sits as the ancestor of *all* objects, in JavaScript it is possible to create objects with no prototype parent!

```
// Empty object that *does* derive from "Object"
let basic = {}
console.log( basic instanceof Object // ⇒ true
           , "toString" in basic)    // ⇒ true

// Empty object that does not derive from "Object"
let maximal = Object.create(null);
console.log( maximal instanceof Object // ⇒ false
           , "toString" in maximal)    // ⇒ false
```

Prototypes let us define properties that are the same for all instances, but properties that differ per instance are stored directly in the objects themselves. E.g., the prototypical person acts as a container for the properties that are shared by all people. An individual person object, like `kathy` below, contains properties that apply only to itself, such as its name, and derives shared properties from its prototype.

```
// An example object prototype
let prototypicalPerson    = {};
prototypicalPerson._world = 0;
prototypicalPerson.speak  = function () {
  console.log(`I am ${this.name}, a ${this.job}, in a world of `
              + `${prototypicalPerson._world} people.`) }
prototypicalPerson.job = `farmer`;

// Example use: Manually ensure the necessary properties are setup
// and then manually increment the number of people in the world.
let person = Object.create(prototypicalPerson);
person.name = `jasim`;
prototypicalPerson._world++;
person.speak() // ⇒ I am jasim, a farmer, in a world of 1 people.

// Another person requires just as much setup
let kathy = { ...prototypicalPerson }; // Same as "Object.create(···)"
kathy.name = `kathy`;
```

```
prototypicalPerson._world++;
kathy.speak() // ⇒ I am kathy, a farmer, in a world of 2 people.
```

Classes are prototypes along with constructor functions!

A *class* defines the shape of a kind of object; i.e., what properties it has; e.g., a Person can `speak`, as all people can, but should have its own `name` property to speak of. This idea is realised as a prototype along with a *constructor* function that ensures an instance object not only derives from the proper prototype but also ensures it, itself, has the properties that instances of the class are supposed to have.

```
let prototypicalPerson    = {};
prototypicalPerson._world = 0;
prototypicalPerson.speak  = function () {
  console.log(`I am ${this.name}, a ${this.job}, in a world of `
              + `${prototypicalPerson._world} people.`) }

function makePerson(name, job = `farmer`) {
  let person  = Object.create(prototypicalPerson);
  person.name = name;
  person.job  = job;
  prototypicalPerson._world++;
  return person;
}

// Example use
let jasim = makePerson(`jasim`);
jasim.speak() // I am jasim, a farmer, in a world of 1 people.
makePerson(`kathy`).speak() // I am kathy, a farmer, in a world of 2 people.
```

```
I am jasim, a farmer, in a world of 1 people.
I am kathy, a farmer, in a world of 2 people.
```

We can fuse these under one name by making the prototype a part of the constructor.

- ◇ By convention, the names of constructors are capitalised so that they can easily be distinguished from other functions.

```
function Person(name, job = `farmer`) {
 this.name = name;
 this.job  = job;
 Person.prototype._world++;
}

Person.prototype._world = 0;
Person.prototype.speak = function () {
  console.log(`I am ${this.name}, a ${this.job}, in a world of `
              + `${Person.prototype._world} people.`) }

// Example use
let jasim = Object.create(Person.prototype)
Person.call(jasim, `jasim`)
jasim.speak() // ⇒ I am jasim, a farmer, in a world of 1 people.

// Example using shorthand
let kasim = new Person (`kathy`)
kasim.speak()  // ⇒ I am kathy, a farmer, in a world of 2 people.
```

```
I am jasim, a farmer, in a world of 1 people.
I am kathy, a farmer, in a world of 2 people.
```

If you put the keyword `new` in front of a function call, the function is treated as a constructor. This means that an object with the right prototype is automatically created, bound to `this` in the function, and returned at the end of the function.

```
  new f(args)
≈ (_ => let THIS = Object.create(f.prototype);
        f.call(THIS, args); return THIS;) ()
```

All functions automatically get a property named `prototype`, which by default holds a plain, empty object that derives from `Object.prototype`. You can overwrite it with a new object if you want. Or you can add properties to the existing object, as the example does.

Notice that the `Person` object *derives* from `Function.prototype`, but also has a *property* named `prototype` which is used for instances created through it.

```
console.log( Object.getPrototypeOf(Person) == Function.prototype
           , Person instanceof Function
```

```
          , jasim  instanceof Person
          , Object.getPrototypeOf(jasim) == Person.prototype)
```

Hence, we can update our motto:

Classes are constructor functions with a prototype property!

Rather than declaring a constructor, *then* attaching properties to its prototype, we may perform both steps together using `class` notation shorthand.

```
class Person {
  static #world = 0
  constructor(name, job = 'farmer') {
    this.name = name;
    this.job  = job;
    Person.#world++;
  }
  speak() {
    console.log('I am ${this.name}, a ${this.job}, in a world of '
                + '${Person.#world} people.')
  }
}

// Example use

let jasim = new Person('jasim')
jasim.speak()
// ⇒ I am jasim, a farmer, in a world of 1 people.

new Person('kathy').speak()
// ⇒ I am kathy, a farmer, in a world of 2 people.
```

```
I am jasim, a farmer, in a world of 1 people.
I am kathy, a farmer, in a world of 2 people.
```

Notice that there is a special function named `constructor` which is bound to the class name, `Person`, outside the class. The remainder of the class declarations are bound to the constructor's prototype. Thus, the earlier class declaration is equivalent to the constructor definition from the previous section. It just looks nicer.

- ◇ Actually, this is even better: The `static #world = 0` declaration makes the property `world` *private*, completely inaccessible from the outside the class. The `static` keyword attaches the name not to particular instances (`this`) but rather to the constructor/class name (`Person`).
- ◇ Indeed, in the previous examples we could have accidentally messed-up our world count. Now, we get an error if we write `Person.#world` outside of the class.

The Iterator Interface

The object given to a `for/of` loop is expected to be iterable. This means it has a method named `Symbol.iterator`. When called, that method should return an object that provides a second interface, the iterator. This is the actual thing that iterates. It has a `next` method that returns the next result. That result should be an object with a `value` property that provides the next value, if there is one, and a `done` property, which should be true when there are no more results and false otherwise.

Let's make an iterable to traverse expression trees.

```
class Expr { // [0] Our type of expression trees
  static Constant(x) {
     let e = new Expr();
     e.tag = 'constant', e.constant = x;
     return e;
     }

  static Plus(l, r) {
     let e = new Expr();
     e.tag = 'plus', e.left = l, e.right = r;
     return e;
     }
}

// [1] The class tracks the progress of iterating over an expression tree
class ExprIterator {
  constructor(expr) { this.expr = expr; this.unvisited = [{expr, depth: 0}]; }
  next () {
   if(this.unvisited.length == 0) return {done: true};
   let {expr , depth} = this.unvisited.pop();
   if (expr.tag == 'constant') return {value: {num: expr.constant, depth}}
```

```
    if (expr.tag == 'plus') {
      // pre-order traversal
      this.unvisited.push({expr: expr.right, depth: depth + 1})
      this.unvisited.push({expr: expr.left, depth: depth + 1})
      }
    return this.next()
  }
}

// [2] We can add the iterator after-the-fact rather than within the Expr class.
Expr.prototype[Symbol.iterator] = function () { return new ExprIterator(this) }

// [3] Here's some helpers and an example.
let num = (i) => Expr.Constant(i)
let sum = (l, r) => Expr.Plus(l, r)
// test ≈ 1 +  (2 + (3 + 4))
let test = sum( num(1), sum( num(2), sum(num(3), num(4))))
// console.log(test) //  ⇒  Nice looking tree ^_^

// [4] We now loop over an expression with for/of
for (let {num, depth} of test)
  console.log(`${num} is ${depth} deep in the expression`)
```

Recall that inside a class declaration, methods that have `static` written before their name are **stored on** the constructor. It appears that static properties are shared by all instances, because the constructor *object* has these as properties rather than particular instance objects.

JavaScript and the Browser

Browsers run JavaScript programs, which may be dangerous and so browsers limit what a program may do —e.g., it cannot look at your files.

An HTML document is a nested sequence of tagged items, which may be interpreted as a living data-structure —with the screen reflecting any modifications.

- ◇ The most important HTML tag is `<script>`. This tag allows us to include a piece of JavaScript in a document.

The data-structure is called the **Document Object Model**, or *DOM*, and it is accessed with the variable `document`.

- ◇ The DOM interface wasn't designed for just JavaScript; e.g., it can be used with XML.

Call the following snippet `test.html`, then open it in your favourite browser.

```
<title> Ola! </title>

<h3 id="myHeader"></h3>

<script>
alert(`Welcome to my webapge!`);

let count = 0;
function modifyThePage(){
  document.title = `New zany title ~ ${Math.random()}`;
  myHeader.innerText = `New zany heading ~ ${count}`;
  count += Math.floor(Math.random() * 10);
}
</script>

<button onclick="modifyThePage();">Change the title and header </button>
```

Such a script will run as soon as its `<script>` tag is encountered while the browser reads the HTML. This page will pop up a dialog when opened to show a message.

Some attributes can also contain a JavaScript program. The `<button>` tag shows up as a button and has an `onclick` attribute whose value (function) will be run whenever the button is clicked.

Notice that by providing ID's to tags, we may refer to them in our JavaScript code.

Including large programs directly in HTML documents is often impractical. The `<script>` tag can be given a `src` attribute to fetch a script file (a text file containing a JavaScript program) from a URL.

```
<h1>Testing alert</h1>
<script src="code/hello.js"></script>
```

The `code/hello.js` file included here contains the simple program `alert("hello!")`.

Reads

- https://eloquentjavascript.net/

  This is a book about JavaScript, programming, and the wonders of the digital.

  Many of the examples in this cheatsheet were taken from this excellent read!
- https://exploringjs.com/index.html

  Exploring JS: Free JavaScript books for programmers —E.g., "JavaScript for impatient programmers"
- https://www.w3schools.com/js/

  This tutorial will teach you JavaScript from basic to advanced.

  Other bite-sized lessions can be found at: https://masteringjs.io/fundamentals
- https://learnxinyminutes.com/docs/javascript/

  Take a whirlwind tour of your next favorite language. Community-driven!
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference

  The JavaScript reference serves as a repository of facts about the JavaScript language. The entire language is described here in detail.
- https://github.com/you-dont-need/You-Dont-Need-Loops

  Avoid The One-off Problem, Infinite Loops, Statefulness and Hidden intent.

Some Fun Stuff ^_^

```
// A "quine" is a program that prints itself, such as this one:
f = _ => console.log(`f = ${f};f()`); f()
// Prints:
// f = _ => console.log(`f = ${f};f()`);f()

// Range of numbers. Including start, excluding end.
let range = (start, end) => [...Array(end - start).keys()].map(x => x + start)
console.log(range(3, 8)) // ⇒ [ 3, 4, 5, 6, 7 ]

// Flatten an array
let xss     = [[1, 2, 3], [4, 5, 6]]
let flatten = xss => [].concat(...xss)
console.log(flatten(xss)) // ⇒ [ 1, 2, 3, 4, 5, 6 ]

// Randomise the elements of an array
let shuffle = (arr) => arr.slice().sort(() => Math.random() - 0.5)
let xs = [1, 2, 3, 4, 5, 6]
console.log(shuffle(xs)) // ⇒ [ 5, 1, 4, 6, 2, 3 ]
```