{POWER.CODERS}

Introduction to JavaScript

Contents

- Small history lesson
- Data Types
- Variables
- Expressions
- Conditions
- Loops
- Functions
- Arrays
- Objects
- Using the console
- ES6

{POWER.CODERS}

Small history lesson

Small history lesson

- Netscape invented JavaScript 1995 to **add actions** to the web and make their browser superior
- Other browsers included their own versions of JavaScript
- The result was chaos, there was **no standard**
- Web developers had a hard time to make sure their web sites were **browser compatible**

Small history lesson

- **ECMAScript** (ES) was created at first standard in 1997
- The current version is ES2024 (15th edition), but the browsers lag behind. ES6 (2015) is the one all modern browsers understand.
- Browsers implement specific features of newer ES versions all the time. Use Caniuse or these comparism tables to check for support.

What is Programming?

- Programming is the art of breaking a problem down into smaller problems.
- Finding the solutions to these smaller problems.
- Putting these pieces back together.

Building blocks of Programming

- Data Types
- Variables
- Conditions
- Loops
- Functions

Data Types

Data types

Indicate in JavaScript what types of value are possible and which kind of operations.

- Number `3.141502653589793`
- String `"Hello world"`
- Boolean `true`
- Undefined
- Null
- Arrays
- Object

How to write strings

```
'This is a string' // single quotes are mostly used
"This is also a string" // double quotes can be used, but should be avoided

'What's up?' // This is an error
'What\s up?' // backslash to "escape" the quotes
"What's up?" //legit use of double quotes

"They said \"What's up?\"."   // more quote escaping
'They said "What\'s up?".'    // ... or use single quotes
```

Print a message

```
console.log('Hello world');
```

Result: 'Hello world'

```
let x = 'This is a ';
x + 'string'; // add string to variable
```

Result: 'This is a string'

```
let y = 1;
x + y;
```

Result: 'This is a 1'

Variables

What is a variable?

In variables you store data

```
//This is a comment. It won't get read by your browser.
//The next line declares a variable.
let x = 10;
```

- The keyword **let** defines a variable.
- **x** is the name of the variable.
- **10** is the value of the variable.
- The **equal sign (=)** is called the assignment operator.

How to declare a variable?

`var` defines a variable globally or locally to an entire function.

```
var x;
```

`let` defines a variable limited in scope to the block, statement or expression in which it is used.

```
let x;
```

`const` defines an **immutable** variable in the same scope than **let**.

```
const x;
```

How do you define a variable?

You will need

- a keyword `let`, `var` or `const`
- a name for the variable (remember the list of forbidden words)
- a value for the variable (optional)

Variables and Data Types

```
/* Multi-line comments are possible as well.
You can write any number you want. */
const pi = 3.141502653589793;

let name = 'Powercoders';
//This is called a string. It is in single or double quotes.

let bool = true;
//This is a Boolean.
```

DESTRUCTURED

Which variable keywords would be better than `var`?

```
var person = {
  firstName : "John",
  lastName  : "Doe",
  age       : 50,
  eyeColor  : "blue"
};

var firstName = person.firstName;
var lastName = person.lastName;
var age = person.age;
var eyeColor = person.eyeColor;
```

Destructured

```
const person = {
  firstName : "John",
  lastName  : "Doe",
  age       : 50,
  eyeColor  : "blue"
};

let {lastName} = person;
const {firstName, age, eyeColor} = person;
```

What is scope?

In JavaScript each function creates a new scope.
Scope determines the accessibility (visibility) of the declared variables.

Variables defined inside a function are not accessible (visible) from outside the function.

GLOBAL SCOPE

You are in the **global scope**, also called **root scope** by default if you are using JavaScript, the **window object**.

GLOBAL VARIABLES

A **global variable** is short for a variable defined in global scope.

Variables declared **outside** of any functions are **global**.

A global variable can be used anywhere **after** its declaration.

Declare all global variables at the top of your JS file.

Local scope

If you declare a variable inside a code block (e.g. function), you create a **local scope**, also called **child scope** or **function scope**

If you have variables inside your local scope, which are not declared there, it will check the global scope for the variable.

Scoping rules on variables were always very confusing and the reason for many bugs.

`let` vs. `var`

- `var` is function-scoped. Every variable declared inside the function is only accessible inside the function.
- `let` is block-scoped (block is anything surrounded by {}). Every variable declared inside a {} block is only accessible inside that block. `const` as well.

Best practice: use `let` over `var`. Use `const` for variables which do not change.

© 2024 Powercoders

What is the value of a in the alert?

```
function q1() {
  var a = 5;
  if(a > 1) {
      a = 3;
  }
  alert(a);
}
```

What is the value of b in the alert?

```
var b = 0;
function q2() {
  b = 5;
}

function q22() {
  alert(b);
}
```

What is the value of a in the alert?

```
function q3() {
  window.a = "hello";
}
function q32() {
  alert(a);
}
```

What is the value of b in the alert?

```
var b = 1;
function q4() {
    var b = "test";
    alert(b);
}
```

What is the value of c in the alert?

```
var c = 2;
if (true) {
    var c = 5;
    alert(c);
}
alert(c);
```

What is the value of c in the alert?

```
let c = 2;
if (true) {
    let c = 5;
    alert(c);
}
alert(c);
```

NAMING OF VARIABLES

JavaScript is **case sensitive**.

```
// These are two different variables
let name = 'Powercoders';
let Name = 'Powercoders';
```

- The first character **must be** a letter, an underscore (_) or a dollar sign ($)
- Numbers are **not allowed** as first character
- Variable names cannot include **mathematical or logical operators**, for instance (*) or (+)
- JavaScript variables **must not contain spaces**.
- Avoid reserved words: List on w3schools

VARIABLES

A variable can be declared without value.

```
let my_variable;
```

A variable declared without value is **undefined**.

The data type `undefined` means the variable has not been assigned.

PROMPT()

To get user input you can use `prompt()`.

```
let name = prompt('What is your name?');
console.log('Hello ' + name);
```

© 2024 Powercoders

Expressions

Expressions

Operator	Operation	Example	Result
`+`	Addition	`2 + 2`	`4`
`-`	Subtraction	`2 - 2`	`0`
`/`	Division	`3 / 2`	`1.5`
`*`	Multiplication	`5 * 2`	`10`
`%`	Remainder / modulus	`9 % 2`	`1`
		`8 % 2`	`0`
`**`	Exponention (x^y)	`2**3`	`8`

Increment & Decrement

to add or substract 1 from a number.

Operator	Example	Result
`val++`	`let a=0, b=10;` `let a=`**`b++`**`;`	a=10 / b=11
`++val`	`let a=0, b=10;` `let a=`**`++b`**`;`	a=11 / b=11
`val--`	`let a=0, b=10;` `let a=`**`b--`**`;`	a=10 / b=9
`--val`	`let a=0, b=10;` `let a=`**`--b`**`;`	a=9 / b=9

ASSIGNMENT OPERATORS

Operator	Example	is equivalent to
`=`	`x = y`	`x = y`
`+=`	`x += y`	`x = x + y`
`-=`	`x -= y`	`x = x - y`
`*=`	`x *= y`	`x = x * y`
`/=`	`x /= y`	`x = x / y`
`%=`	`x %= y`	`x = x % y`

Online resources

- JavaScript Operator lookup
- MDN JavaScript Grammar & Types
- MDN JavaScript basics
- JavaScript introduction
- JavaScript tutorial
- Using variables in JavaScript

{POWER.CODERS}

Conditions

Conditions

With **conditions** you can influence the code the browser will execute.

- If a condition is **true** we will do something.
- If a condition is **false** we won't.

If the teacher is sitting in front of the laptop she will go to the next slide now.

Conditions

In JavaScript a condition looks like this:

```
if (statement) {
  doSomething();
}
```

Only if the statement is true `doSomething` will be executed.

Boolean logic

Booleans are used by JavaScript to **check if a statement is true**.

Logical operators allow you to connect as many expressions as you wish.

Operator	Description	Example
`&&`	logical **AND**	true && true = true
`||`	logical **OR**	true || false = true
`!`	logical **NOT**	!false = true

Logical operators

True and **False** are often represented by 1 and 0.

A	B	A `&&` B	A `||` B	`!` A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Comparison Operators

Are used in logical statements to determine equility or difference between variables or values. They return **true** or **false**.

Operator	Description	let x=5;	Result
==	equal to	x == '5'	True
===	equal value and equal type	x === '5'	False
!=	not equal	x != 3	True
>	greater than	x > 4	True
<	less than	x < 4	False
>=	greater than or equal to	x>=5	True
<=	less than or equal to	x<=4	False

`if else`

The `else` statement tells JavaScript to execute something if the condition is **false**.

```
if(statement){
  doSomething();
} else {
  doSomethingElse();
}
```

If the condition is **false** `doSomethingElse` will be executed.

Ternary condition

```
condition ? expr1 : expr2
```

If the condition is true, provide expr1 else provide expr2.

It is like shorthand for an `if` `else` statement.

Ternary condition

```
let age = 21;
let result;

if(age >= 20){
  result = "User can view content";
} else {
  result = "User cannot view content";
}
console.log(result);
```

```
let age = 21;
let result = age >= 20 ? "User can view content" : "User cannot view content";
console.log(result);
```

`switch`

In case of many different conditions resulting in different actions use `switch` instead of multiple `if/else if` statements.

```
switch(expression){
  case x:
    // code block
  break;
  case y:
    // code block
  break;
  default:
    // code block
}
```

- `default` gets executed if no case matches. **Always** have it.
- `break` will stop the execution of more code inside the `switch` block.

`switch` vs `if else if`

- **Testing expression:** `if else if` can test expressions based on conditions, `switch` only on a single integer or string
- **Multi-way branching:** `switch` is faster if there is a large group of values to select among
- **Boolean values:** `if else if` is better for conditions resulting in true or false

QUESTION ABOUT SWITCH

```
function moveCommand(direction) {
    var whatHappens;
    switch (direction) {
        case "forward":
            break;
            whatHappens = "you encounter a monster";
        case "back":
            whatHappens = "you arrived home";
            break;
            break;
        case "right":
            return whatHappens = "you found a river";
        case "left":
```

- Return value of `moveCommand("forward")` **undefined**
- Return value of `moveCommand("back")` **you arrived home**
- Return value of `moveCommand("right")` **you found a river**

{POWER.CODERS}

Loops

Loops

So far our code has been linear, read line after line. **Loops** allow you to run the same code **repeatedly**, adding a different value each time.

There are several types of loops:

- `while`
- `for`
- `forEach`
- `do while`
- `for of` covered here
- `for in` covered here

`while`

The `while` loop is the easiest. It repeats through a block of code, as long as a specified condition is **true**.

```
while(statement){
  //codeblock this gets looped until the statment is false
}
```

If a condition is always **true**, the loop will run **forever**.
Make sure that the condition eventually becomes **false**.

`for`

The `for` loop is most commonly used. It has 3 statements, separated by **semi-colons**:

- **initialization**: declare a new variable used for counting the loops, e.g. `x=0`
- **condition**: as long as the condition is true the loop will run, e.g. `x<=5`
- **iterator**: is executed after every iteration of the loop, e.g. `x++`

```
for(initialization; condition; iterator){
  //codeblock this gets looped until the condition is false
}
```

`do while`

The `do while` loop will always be executed **at least once**, even if the condition is false.

```
do { //code block here } while(statment);
```

Controlling loop executions

The `break` statement jumps out of a loop and continues executing the code **after** the loop.

The `return` keyword will also break the loop as it immediately returns some value from a loop inside a function.

The `continue` statement breaks **only one iteration** in the loop and continues with the **next** iteration.

{POWER.CODERS}

Functions

What is a function?

A block of code designed to perform actions. The purpose is to perform a particular task (one piece of solving the problem).

- **Code reuse**: define the code once and use it many times
- **Different arguments**: used with the same code will produce different results.

Functions we know

- `window.alert()`
- `window.prompt()`
- `console.log()`

See the () at the end. You need these to execute the function.

Function declaration

```
function myFunction(param1, param2, param3){
return param1 + param2 * param3;
}
```

- `function` keyword to tell JS that a function declaration starts.
- `name` of the function the same rules apply as in naming variables.
- `(param1, param2, param3)` directly after the **name** for (optional) **arguments**. The "placeholders" are called parameters.

  You can have as many as you wish.
- `{}` holds the code to be executed when the function is called.
- `return` keyword (optional) breaks the execution and returns the value.
- `param1 + param2 * param3` is called the **return value**.

  You can only have **one** return value.

Calling a function

To execute the code inside a function, you need to call it.

```
let arg1 = 1;
let arg2 = 2;
let arg3 = 3;

myFunction(arg1, arg2, arg3);
```

- `name` - start by writing the name of the function
- `(arguments)` - add the arguments in parentheses
- `;` - always remember to use a semicolon

Calling a function

When calling a function, provide the arguments in the same order in which you defined them.

- If there a fewer arguments when defined, they will be declared without a value and get the datatype **undefined**.
- If there are more, they will be added inside the array **arguments**.

Calling a function

Once the function is declared, you can call it **everywhere** as many times as you want:

- before the function declaration
- after the function declaration
- inside the function declaration (=recursive functions)

Default arguments

You can give default arguments when declaring a function.

```
let myFunction = function(arg1=5, arg2=3, arg3=1){
return arg1 + arg2 * arg3;
}

myFunction();
myFunction(20);
```

Function Expression

We said that we can declare a function like this:

```
function myFunction(arg1, arg2, arg3) {
      return arg1 + arg2 * arg3;
}
```

But a function can also be defined using an **expression**:

```
const myFunction = function(arg1, arg2, arg3) {
      return arg1 + arg2 * arg3;
}
```

Function Expression

- A function expression can be stored in a variable with the keyword `const`.
- After a function expression has been stored in a variable, it can be referenced (called) by it.
- Functions stored in variables do not need function names. They are called **anonymous** functions.

DECLARATION VS. EXPRESSION

- Function **declarations** load **before** any code is executed.
- Function **expressions** load only when the interpreter reaches **that line** of code.

Function expressions become sometimes more useful than function declarations, e.g. if they are used as arguments to other functions.

Arrow function

```
(parameters) => {statements}
```

- An arrow function is a compact alternative.
- An arrow function cannot be used as a method.

COMPARING FUNCTIONS

Breakdown to get an arrow function

```
function (a){
    return a + 100;
  }

  /**
  * 1. Remove the word "function" and place arrow between
  * the argument and opening body bracket */
  (a) => {
    return a + 100;
  }
```

COMPARING FUNCTIONS

Breakdown to get an arrow function

```
/**
  * 1. Remove the word "function" and place arrow between
  * the argument and opening body bracket */
  (a) => {
    return a + 100;
  }

  /**
  * 2. Remove the body brackets and word "return" -- the return is implied.
  */
  (a) => a + 100;
```

Comparing functions

Breakdown to get an arrow function

```
/**
  * 2. Remove the body brackets and word "return" -- the return is implied.
  */
  (a) => a + 100;

  /**
  * 3. Remove the argument parentheses
  */
  a => a + 100;
```

ARROW FUNCTIONS

Each step results in a valid arrow function

- If there are no parameters: `() => { statements }`
- If there is 1 parameter: `parameter => { statements }`
- If an expression is returned: `parameter => expression`

No parameters

```
() => {
    let x = 5 + 100;
    alert(x);
  }
```

1 PARAMATER

```
a => {
    let x = a + 100;
    alert(x);
  }
```

More paramaters

```
(a, b) => {
    let x = a + b;
    alert(x);
  }
```

Expression

```
(a, b) => a + b;
```

Recursive functions

A recursive function calls itself inside its code block.

```
let countdown = function(value) {
if (value > 0) {
console.log(value);
return countdown(value - 1);
} else {
return value;
}
};
countdown(10);
```

ONLINE RESOURCES

- Arrow functions
- When to use arrow functions

{POWER.CODERS}

Arrays

What is an array

Arrays store multiple values in a single variable.

```
let topics = ["HTML","CSS","JS"];
```

- An array is a **special type of object**.
- An array is a collection of often similar data.
- The items in an array have a guaranteed order.
- An array can hold any data in JS: objects, numbers, strings ...

ACCESSING AN ARRAY

You access an array element by referring to the **index number** written in **square brackets**.

```
let firstTopic = topics[0];

  topics[2] = "jQuery";
  // Possible to overwrite a value in an array
```

- Array indexes start with **0**
- Referring to an index outside of the array, returns **undefined**
- An array uses **numbers** to access its elements, an object uses **names**

© 2024 Powercoders

CREATING ARRAYS

There are several ways how to declare an array

```
let topics = new Array(3);
  topics[0] = "HTML";
  topics[1] = "CSS";
  topics[2] = "JS";
```

```
let topics = new Array(); // more dynamic without argument
  topics[0] = "HTML";
  topics[1] = "CSS";
  topics[2] = "JS";
  topics[3] = "PHP";
```

```
let topics = ["HTML","CSS","JS"] // recommended way to declare arrays
```

`length`

- Remember: `length` is a built-in JS property of any object.
- `length` returns the number of items inside an array.
- The value of `length` is always one more than the highest index.
- If the array is empty, the `length` property returns **0**.

Multidimensional Array

In Javascript a multidimensional array is an array where each element is also an array.

```
let timeSpent = [
          ['Work', 9],
          ['Eat', 2],
          ['Commute', 1],
          ['Watch TV', 2],
          ['Sleep', 7]
      ];

      console.log(timeSpent[0][1]);
```

To access an element of the multidimensional array, you first use square brackets to access an element of the outer array which returns an inner array; and then use another square bracket to access the element of the inner array.

Looping with `for...of`

The `for...of` loop is intended for iterating over arrays, providing a simpler alternative to `forEach`.

```
let list = ["doors", "windows", "rooms"];
        for (let item of list) {
            console.log(item);
        }
```

It also works for all iterable objects, including strings:

```
for (let char of "hello") {
            console.log(char);
        }
```

Looping with `forEach`

The `forEach` method, introduced in ES5, is easier to read than the `for` loop and prevents accidental breaking of the loop.

```
list.forEach(function(value, index) {
            // Code block here
            console.log(value, index);
        });
```

However, if you need to `return` a value and break the loop, `forEach` is not suitable.

Using `forEach` with arrow Functions (ES6)

In ES6, you can use arrow functions with `forEach` for a more concise syntax:

```
list.forEach((element, index) => {
            // More code here
            console.log(`Rank ${index + 1}: competitor: ${element}`);
        });
```

Combining Arrays with `concat`

The `concat` method merges two arrays into **one new array**:

```
const t1 = ["HTML", "CSS"];
        const t2 = ["JS", "PHP"];
        const topics = t1.concat(t2);
        console.log(topics); // ["HTML", "CSS", "JS", "PHP"]
```

Array methods

Next to `concat` these are some of the most important methods you can use:

- `topics.toString()` This will return all the elements as a string seperated by a comma.
- `topics.push("MySQL")` This will add "MySQL" at the end of the array.
- `topics.pop()` This will return the last element of the array and will remove it.

Array methods

- `topics.shift()` This will return the first element of the array and will remove it.
- `topics.sort()` This will sort the array **alphabetically**.

and many more

array.filter()

```
//the old way
  function WhoIsMe(){
    let names = ["Susanne","Christina","Linus","Hany"];
    for(let name of names){
      if(name==="Susanne")
        return "Susanne";
    }
  }
```

```
//the new way
  let me = names.filter(name => {
    return name==="Susanne";
  });
```

`array.map()`

```
//the old way
  let users = [{
    name: "Susanne",
    surname: 'Koenig',
    age: 88
  },{
    name: "Andrina",
    surname: 'Beuggert',
    age: 88
  }];
  let allSurnames = [];
  for(let user of users){
    allSurnames.push(user.surname);
  }
```

```
//the new way
  let allSurnames = names.map(user => user.surname)
```

array.reduce()

```
//the old way
  let cart = [
  {
      product_id: 89898,
      product_name: 'Napkin red',
      product_price: 6.50,
      quantity: 10
  },
  {
      product_id: 123,
      product_name: 'Plastic forks',
      product_price: 1.25,
      quantity: 15
  }
  ];
  let total = 0;
```

```
//the new way
  let total = cart.reduce((acc, current) =>
    acc += current.product_price * current.quantity,0
  );
```

Array methods

■■■■ .map(■→●) → ●●●●

■■●■ .filter(■) → ■■■

●●■■ .find(■) → ■

●●●■ .findIndexOf(■) → 3

■■■■ .fill(1, ●) → ■●●●

●■■● .some(■) → true

■■■● .every(■) → false

Online resources

- filter() method
- map() method
- Use map(), reduce() and filter()
- Array methods explained

{POWER.CODERS}

Objects

What is an object

An **Object** can contain **many** values and help organize and structure them.

An **Object** is a data type (like numbers or strings), but also a **data structure**.

An **Object** is a collection of data types. They can even have methods (=functions) in them.

Think of a collection as a list of values that are written as **key:value** pairs.

An example

```
const person = {
name: "John",
age: 38,
eyeColor: "green",
isMarried: false
};
```

- Each line, separated by a comma, is called a **property**.
- The names of the variables on the left are called **property keys**.
- The values on the right side are called **property values**.

Javascript objects are containers for named values.

Interact with Objects

There are 2 ways to access an **object property**:

```
person.age;
person["age"];
```

This is how to access an **object method**:

```
person.hello();
    // if you type person.hello without () you get back the defintion
```

ANOTHER EXAMPLE

```
console.log(person.name.length);
```

The **log()** function is actually a method of the **console** object.

The built-in **length** property is used to count the number of characters.

`null`

You can empty an object by setting it to null.

```
person = null;
    person.name = "Susanne"; // not possible
```

Remember

```
const person = {
      name: "John",
      age: 38,
      isMarried: false
    };
```

Initializing the object this way, created **one single object**.
But Simon and Marc are persons, too.

We need a more general object type that can be used to create a number of objects of the same type.

Object constructor

```
function Person(name, age, married) {
      this.name = name;
      this.age = age;
      this.isMarried = married;
      this.sayHello = function() {
        return "Hello " + this.name;
      };
    };
```

An object constructor is a function that performs the task of defining an object.

The `this` keyword refers to the **current object**.

You also need `this` to access the variables of your own object, e.g. **inside a method**.

What keyword is used for creating an instance of an object?

Creating instances of an object

Once you have an object constructor, use the `new` keyword to create a new object of the same type called **instance**.

```
const susanne = new Person("Susanne", 41, false);
    const max = new Person("Max", 25, true);

    console.log(susanne.age); // Output: 41
    console.log(max.sayHello()); // Output: Hello Max

    susanne.age = 45; // Possible to change a property value
    console.log(susanne.age); // Output: 45

    susanne.gender = "female"; // Possible to add new property
    console.log(susanne.gender); // Output: female
    delete susanne.gender; // Possible to delete property
```

Classes

```
class Person {
      constructor(name, age, married) {
        this.name = name;
        this.age = age;
        this.isMarried = married;
      }
      hello() {
        return "Hello " + this.name;
      }
    }
```

- Classes are **templates** for JavaScript objects
- A newer notation for the object constructor function

`Object` object

`Object.keys()` lists all property names of an object in an array.

`Object.values()` returns all property values of an object in an array.

```
Object.keys(susanne);
    // ["name", "age", "isMarried", "hello"]
```

Looping through an object

The `for in` loop is intended for iterating over the keys of an object.

```
let obj = {doors: 2, windows: 8, rooms: 5};
    for(let x in obj){
      console.log(x);
    }
```

- Do not use this loop for arrays.
- The iterating variable `x` is always a string.

Looping through an object

Alternatively you can use `Object.keys().forEach()` or `Object.values().forEach` to loop through an object.

```
let obj = {doors: 2, windows: 8, rooms: 5};
    Object.keys(obj).forEach(function(property_name){
      console.log(obj[property_name]);
    });
```

```
let obj = {doors: 2, windows: 8, rooms: 5};
    Object.values(obj).forEach(function(property_value){
      console.log(property_value);
    });
```

Built-in objects

- `Math` to perform mathematical tasks
- `Date` to work with dates

Math

```
let pi = Math.PI;
    console.log(Math.floor(pi));
    console.log(Math.round(pi));
    console.log(Math.ceil(pi));

    let randomNumber = Math.ceil(Math.random() * 10);
    console.log(randomNumber);
```

Date

```
function printTime(){
      let currentDate = new Date();
      let hours = currentDate.getHours();
      let mins = currentDate.getMinutes();
      let secs = currentDate.getSeconds();

      console.log(hours + ":" + mins + ":" + secs +"\n");
    }

    setInterval(printTime, 1000); // prints current time each second
```

Dynamic object properties

With `[]` in your property name, you can put in dynamic values, like another variable or a calculation.

```
const name = "first name";

  const obj = {
    [name]: "Susanne",
    [5 + 13]: 38,
    experience: 13
  }
```

Online resources

Check what we learned so far on w3schools.com and mdn.com, e.g.

- functions
- objects
- arrays
- Data structures
- `this` keyword in JavaScript
- Alternative tutorial to explain `this`

Using the console

USE OF THE CONSOLE

Dev Tools include a REPL (read, evaluate, print, loop), better known as console.

```
console.log('Hello ' + 'world');
```

Result: 'Hello world'

```
let x = 'This is a ';
  x + 'string'; // add string to variable
```

Result: 'This is a string'

```
let y = 1;
  x + y;
```

Result: 'This is a 1'

Console cheatsheet

- `clear()`: clears the console
- `alert()`: writes output in a popup
- `prompt()`: asks for input in a popup
- `Number()`: converts a string to a number
- The arrow key up shows you a history of your console entries
- Shift + Enter creates a new line without executing it

{POWER.CODERS}

ES6

Prototypal inheritance

Inheritance is a way to share common logic in programming

```
class Animal {
    constructor(name) {
      this.name = name;
    }
    jump() { console.log(`${this.name} is jumping.`); }
  }
  class Bird extends Animal {
    fly() { console.log(`${this.name} is flying.`); }
  }
  class Dog extends Animal {
    bark() { console.log(`${this.name} says "Woof!"`); }
  }
```

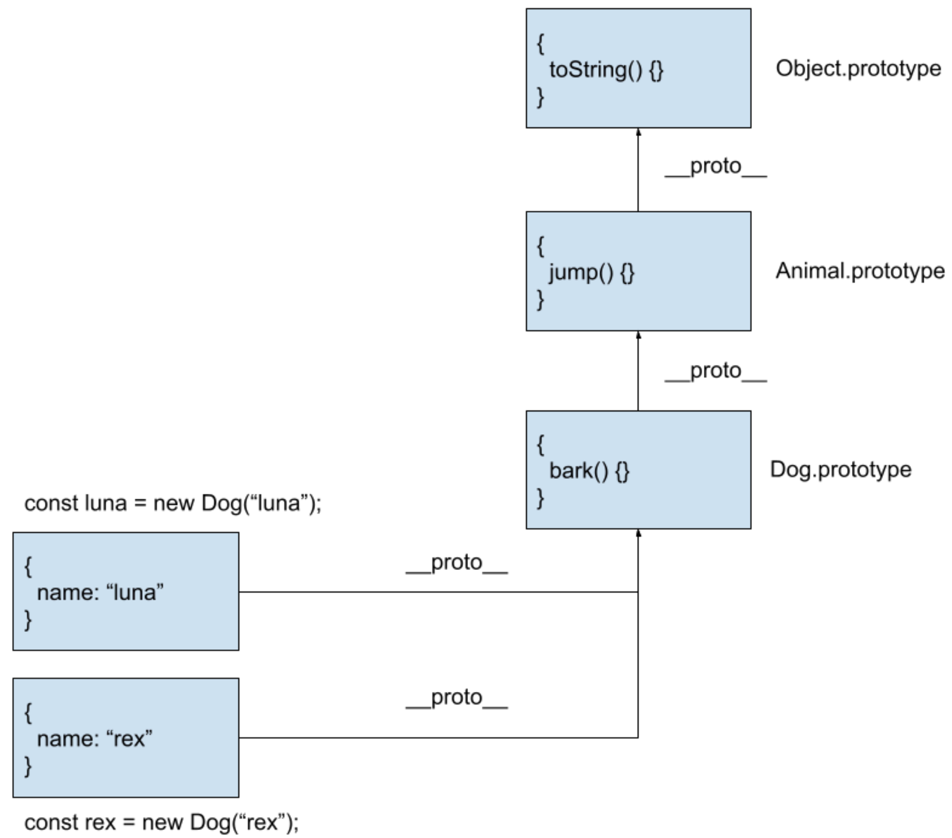
PROTOTYPAL INHERITANCE

```
const myDog = new Dog("Luna");
  console.log(myDog.name); // "Luna"
  myDog.jump(); // "Luna is jumping."
  myDog.bark(); // "Luna says 'Woof!'"
  myDog.fly(); // Uncaught TypeError: myDog.fly is not a function
```

However, we don't see jump or bark defined on the object:

```
> myDog
<· ▼Dog {name: 'Luna'} i
      name: "Luna"
    ▶ [[Prototype]]: Animal
```

PROTOTYPE CHAIN



Prototype chain

- **Data** is **unique** to every object (e.g. the animal name)
- **Methods** are **shared** logic (e.g. `jump()`)
- Shared logic is stored on a shared object (**prototype object**)
- Each object has a **link** to this prototype object (**__proto__**)
- When a method is called
  - Is is defined on the current object? If yes call it
  - If no, is the method defined on the prototype object?
  - If no, go up until the prototype object is null. Then the method is not defined at all