

Internet of Things (IOT)

IoT Programming

Davide Ancona
DIBRIS - Università di Genova

a.y. 2024-2025

Main references



David Flanagan. [JavaScript: The Definitive Guide, 7th Edition](#). O'Reilly Media, 2020



Marc Wandschneider. [Learning Node.js, 2nd Edition](#). Addison-Wesley, 2016

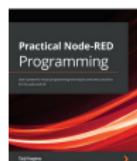
Other suggested books



Douglas Crockford. [JavaScript: The Good Parts](#). O'Reilly Media, 2008



Mario Casciaro and Luciano Mammino. [Node.js Design Patterns, 3rd Edition](#). Packt Publishing, 2020



Taiji Hagino, Nick O'Leary. [Practical Node-RED Programming](#). Packt Publishing, 2021

Historical overview

Names and Versions

- JavaScript was created at Netscape and appeared in late 1995
- a trademark from Sun Microsystems (now Oracle) to describe Netscape (now Mozilla) implementation of the language
- language standardized by ECMA (European Computer Manufacturer's Association) and called **ECMAScript**
- **ECMAScript 5.1** (June 2011): still widespread version
- most recent version: **ECMAScript 15.0** (June 2024)
- specs at [https://262.ecma-international.org/\\$V\\$/](https://262.ecma-international.org/V/)
with $\$V\$ \in \{5.1, 6.0, \dots, 15.0\}$

Lot of ES 5.1 legacy code!

Babel: a useful projects to translate next-gen JS into ES 5.1

The screenshot shows the Babel website homepage. The header features the word "BABEL" in a stylized font. The main title "Babel is a JavaScript compiler." is prominently displayed in yellow. Below it, the tagline "Use next generation JavaScript, today." is shown. A message通知 "Babel 7.21 is released! Please read our blog post for highlights and changelog for more details!" is visible. A central feature is a code editor split into two panes: "Put in next-gen JavaScript" and "Get browser-compatible JavaScript out". Both panes contain the same code: `let yourTurn = "Type some code in here!";`. The footer includes navigation icons and links to "Docs", "Setup", "Try it out", "Videos", "Blog", "Search", "GitHub", and "Team".

Typed JavaScript

- **TypeScript** is a statically typed version of JavaScript
- correct TypeScript programs can be directly translated into JavaScript code
- in fact, TypeScript is a super-set of JavaScript
- its compiler is also useful to translate next-gen JS into ES 5.1

The screenshot shows the official TypeScript website. At the top, there is a navigation bar with links for "TypeScript", "Download", "Docs", "Handbook", "Community", "Playground", and "Tools". The main headline reads "TypeScript is **JavaScript with syntax for types.**". Below this, a sub-headline states: "TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale." A button labeled "Try TypeScript Now" is present, along with links for "Online or via npm". To the right, a code editor window displays the following TypeScript code:

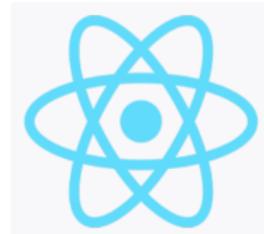
```
const user = {
  firstName: "Angela",
  lastName: "Davis",
  role: "Professor",
}

console.log(user.name)
```

A tooltip or error message appears over the word "name": "Property 'name' does not exist on type '{ firstName: string; lastName: string; role: string; }'".

What is JavaScript

- JavaScript is the most popular scripting language for the Web
 - supported by major Web browsers
 - used in many Web sites
- JavaScript is not used only for Web programming
- with JavaScript it is possible to develop
 - native applications for mobile phones (with different technologies)
 - games (with Unity)
 - servers, services (with Node.js, Node-RED)



Learning JavaScript with Node.js

Focus in this course

Development of IoT middleware and server side applications in Node.js

How to run Node.js script

- the Node shell provides the Read-Eval-Print-Loop (REPL) to test code

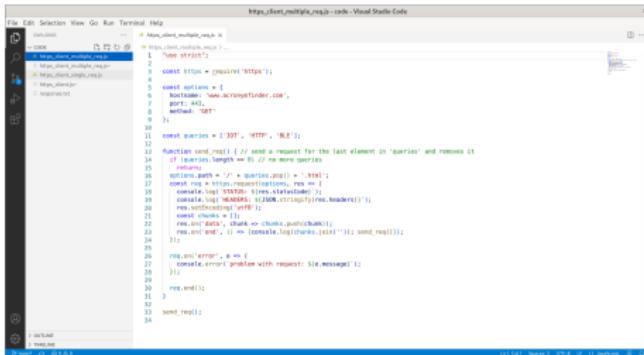
```
> console.log('Hello world!')  
Hello world!  
undefined
```

- scripts can be executed via the command line with node

```
$ node hello.js
```

Developing code in JavaScript

Visual Studio Code

```

File Edit Selection View Go Run Terminal Help https://client.multiple_reqs.in/-code - Visual Studio Code
https://client.multiple_reqs.in/
1 // https://client.multiple_reqs.in/
2
3 const start = 2;
4
5 const https = require('https');
6
7 const options = {
8   hostname: 'www.acronymfinder.com',
9   port: 443,
10   method: 'GET'
11 };
12
13 const queries = ['HTTP', 'HTTPs'];
14
15 function send(next) {
16   if (queries.length === 0) {
17     return next();
18   }
19   const query = queries.pop();
20   const req = https.request(options, res => {
21     console.log(`Received response for ${query}!`);
22     const chunks = [];
23     res.on('data', chunk => chunks.push(chunk));
24     res.on('end', () => console.log(`[${query}] ${res.statusCode}`));
25   });
26
27   req.on('error', e => {
28     console.error(`Problem with request: ${e.message}`);
29   });
30
31   req.end();
32
33   send(next);
34 }

```

- **Visual Studio Code** is one of the most popular IDE for JavaScript
- **freely downloadable** and easily installable
- must be run from the terminal

\$ code .

JavaScript is very flexible, it rarely complains!

Static errors

Only syntax errors are statically (=before execution) detected

```
if x console.log('ok');  
^
```

```
Uncaught SyntaxError: Unexpected identifier 'x'
```

Dynamic errors

Except for syntax, all other errors, if any, are detected at runtime

```
> let a=[1,2,3]; // array
```

```
undefined
```

```
> a[5]; // no error!
```

```
undefined
```

```
> null.length;
```

```
Uncaught TypeError: Cannot read properties of null (reading 'length')
```

JavaScript pitfalls



Strict versus unrestricted mode

- JavaScript is very flexible, it rarely complains
- for safety reasons be sure to **always run** programs in strict mode

```
'use strict'; // your code should always starts in this way
```

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

To run a REPL session in strict mode use the following option:

```
$ node -use-strict
```

JavaScript pitfalls



Use of semicolon

- most times the semicolon symbol `;` is automatically inserted at the end of statements
- to avoid subtle errors, it is **highly recommended** to always insert manually **semicolon at the end** of each statement

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

Types

Primitive values of built-in types

- special values **null** and **undefined**
- numbers (Number/BigInt)
- strings
- booleans
- symbols

Object types

- ordinary objects: references to finite maps (=dictionaries) from **properties keys** to values
- special objects: the global object, arrays, and functions

typeof operator

Examples

```
> typeof null  
'object'  
> typeof undefined  
'undefined'  
> typeof 12  
'number'  
> typeof BigInt(424242424242424242424242)  
'bigint'  
> typeof 'a string'  
'string'  
> typeof true  
'boolean'
```

typeof operator

Examples

```
> typeof {x:1,y:1} // object literal with properties 'x' and 'y'  
'object'  
> typeof [1,2,3] // array literal  
'object'  
> typeof function(){} // expression defining a function  
'function'  
> typeof Symbol() // symbols are used as unique names  
'symbol'
```

null versus undefined

In a nutshell

- **undefined**: uninitialized variable, absent object property or returned value; most statements in the REPL shell return **undefined**
- **null**: variable/property holding no value

Example

```
> let x;  
undefined  
> x = {}; // empty object  
{ }  
> x.missing;  
undefined  
> let y;  
undefined  
> y  
undefined  
> y = null;  
null
```

When JS complains...

Not allowed to access properties of **null/undefined!**

Example

```
> null.x;  
Uncaught TypeError: Cannot read properties of null (reading 'x')  
> undefined.x;  
Uncaught TypeError: Cannot read properties of undefined (reading 'x')
```

Not allowed to call values which are not functions

Example

```
> let a=[1,2,3];  
undefined  
> a(0);  
Uncaught TypeError: a is not a function
```

Constants

Since ECMAScript 6

```
> const PI = 3.141593;  
undefined  
> PI;  
3.141593  
> PI++;  
Uncaught TypeError: Assignment to constant variable.
```

Old style

```
> var PI = 3.141593; // the simplest way!  
undefined  
> console.log(PI);  
3.141593  
undefined  
> PI++; // this is not really a constant!  
3.141593  
> console.log(PI);  
4.141593  
undefined
```

Numbers

All numbers represented with the 64-bit floating-point IEEE 754 standard

Integers

inclusive integer interval:

$-9007199254740992 (-2^{53})$ and $9007199254740992 (2^{53})$

```
> 1024 * 1024
1048576
> 1048576
1048576
> 32437893250 + 3824598359235235
3824630797128485
> -38423538295 + 35892583295
-2530955000
```

Beware of approximations!

```
> 1024 == 1024+0.1-0.1
false
> 1024+0.1-0.1
1023.999999999999
```

Arithmetic operators

Standard operators: `+`, `-`, `*`, `/`, `%`, `**` and predefined object `Math`

```
> Math.pow(2,53)==2**53 && 2**53==9007199254740992
```

```
true
```

```
> Math.round(.6)
```

```
1
```

```
> Math.ceil(.6)
```

```
1
```

```
> Math.floor(.6)
```

```
0
```

```
> Math.abs(-5)
```

```
5
```

```
> Math.sqrt(3)
```

```
1.7320508075688772
```

```
> Math.pow(3, 1/3)
```

```
1.4422495703074083
```

```
> Math.sin(0)
```

```
0
```

```
> Math.log(10)
```

```
2.302585092994046
```

```
> Math.exp(3)
```

```
20.085536923187668
```

Useful conversion operators into strings

Predefined functions of the object Number.prototype

```
> const PI = 3.1415;
undefined
// a string representing PI in fixed-point notation with 2 decimal places
> PI.toFixed(2);
'3.14'
// a string representing PI in exponential notation
> PI.toExponential();
'3.1415e+0'
> const R = 287.05;
undefined
// a string representing R to 4 significant digits
> R.toPrecision(4);
'287.1'
```

Special numerical values

Infinity and NotANumber values

```
> 5 / 0  
Infinity  
> -5 / 0  
-Infinity  
> 0/0  
Nan  
> 5/0 == Infinity  
true
```

Positive and negative zero

```
> 1/0  
Infinity  
> 1/-0  
-Infinity  
> -1/Infinity  
-0  
> 0 == -0  
true  
> 1/0 == 1/-0  
false
```

Useful properties for numbers

Useful predefined functions and constants

- **Infinity** and **NaN**: non writable, non configurable properties of the **global object**
- **parseInt**, **parseFloat** and **isNaN**: global functions (other properties of the **global object**)
- **Number**: global constructor (other property of the **global object**)

Alert: some overlapping between properties of the **global object** and **Number**

Examples

```
> Number.POSITIVE_INFINITY == Infinity
true
> Number.NEGATIVE_INFINITY == - Infinity
true
> Number.MAX_VALUE
1.7976931348623157e+308
> Number.MAX_SAFE_INTEGER
9007199254740991
```

Examples of use of global properties for numbers

Parsing utilities

```
> parseInt("32523523626263");
32523523626263
> parseFloat("82959.248945895");
82959.248945895
> parseInt("234.43634");
234
> parseFloat("10");
10
> parseInt("a number")
NaN
```

Testing numbers

```
> NaN === NaN
false // !!!
> isNaN(NaN) && ! isNaN(42)
true
> isFinite(42)
true
> ! isFinite(Infinity) && ! isFinite(-Infinity) && ! isFinite(NaN)
true
```

More on integer literals

Binary, octal, and hexadecimal literals are supported

```
> 0b110 // binary integer literal  
6  
> 0o10 // octal integer literal since ECMA6  
8  
> 0xf // hexadecimal integer literal  
15
```

Legacy octal literals: beware!

Not allowed in strict mode

```
> 'use strict'; 070;  
Uncaught SyntaxError: Octal literals are not allowed in strict mode.  
  
> 'use strict'; 080;  
Uncaught SyntaxError: Decimals with leading zeros are not allowed in strict mode.
```

Boolean values

In a nutshell

- literals: `false` and `true`
- standard operators: `&&` (and), `||` (or), and `!` (not)

Implicit conversions

All non boolean values are implicitly interpreted as boolean values whenever needed

- falsy values: interpreted as false

```
undefined
null
0
-0
BigInt(0)
NaN
"" or '' // the empty string
```

- truthy values: all other non-boolean values are interpreted as true

Logical operators

and/or are short-circuit operators

```
> false && null.length  
false  
> true || null.length  
true  
> 0 && null.length  
0  
> Infinity || null.length  
Infinity  
> Infinity && 42  
42  
> NaN || 'hi'  
'hi'
```

the result of negation is always a boolean value

```
> !undefined;  
true  
> !1;  
false
```

Idiomatic usages of Boolean values and operators

Examples

```
if (o !== null) ...
if (o) ... // stricter condition: any non-falsy value

// non-zero alternatives
let limit = min_limit || preferences.limit || 42;

// default values for parameter options
options=options||{depth:default_depth}
```

Strings

In a nutshell

- JavaScript strings are sequences of unsigned 16-bit values
- UTF-16 encoding of the Unicode is employed
- strings are **immutable values**
- the type **char** is not supported

String literals and templates

```
> "a string";
'a string'
> 'another string';
'another string'
> 'a double quote ''';
'a double quote '''
> "a single quote ''";
'a single quote \\'''
> console.log('The result is ${6*7}'); // a string template
The result is 42
undefined
```

String properties/functions

Some other standard functions/operators

```
> 'hello'+' '+'world!';
'hello world!'
> let num=42;
undefined
> 'the answer is '+num;
'the answer is 42'
> 4+2+' is the answer';
'6 is the answer'
> 'the answer is '+4+2;
'the answer is 42'
> 'JavaScript'.indexOf('Script');
4
> 'JavaScript'.substr(4,5); // first index (inclusive), length
'Scrip'
> 'javaScript'.slice(4,9); // first index (incl.), last index (excl.)
'Script'
> 'www.ecma-international.org'.split('.');
[ 'www', 'ecma-international', 'org' ]
> '\n hi there\n\n '.trim(); // trims the start and the end
'hi there'
```

From primitive strings to String objects

Primitive strings can be converted in [read-only pseudo-arrays](#)

The predefined `String` constructor

```
> s='hi';
'hi'
> typeof s;
'string'
> s[0]; // implicit conversions
'h'
> s[s.length-1]; // implicit conversions
'i'
> t=new String(s); // explicit conversion
[String: 'hi']
> typeof t;
'object'
> Array.isArray(t);
>false
> t==s; // strict equality comparison, no conversion
>false
> t==s; // loose equality comparison, implicit conversion
>true
```

Strings and pseudo-arrays

Primitive or object strings are always immutable!

```
> let s = 'JavaScript';
undefined
> let t = 'JavaScript';
undefined
> s === t;
true
> s.toUpperCase();
'JAVASCRIPT'
> s === t;
true
> let u = new String(s);
undefined
> u[0]='s'; // in unrestricted mode
's'
> u;
[String: 'JavaScript']
u[0]='s'; // in strict mode
Uncaught TypeError: Cannot assign to read only property '0' of object 'String'
```

Regular expressions

Supported literals and predefined constructor RegExp

```
> pattern = new RegExp('\\d+'); // the use of /\d+/ is preferable
/\d+/
> text = 'date:27/01/2007';
> pattern.test(text);
true
> /\d/.test(text); // prefer reg. exp. literals over RegExp cons.
true
> text.search(pattern);
5
> text.match(pattern);
[ '27', index: 5, input: 'date:27/01/2007', groups: undefined ]
> text.match(/\d/g); // global flag is set
[ '27', '01', '2007' ]
> text.replace(pattern, "#");
'date:#/01/2007'
> text.split(/\d/);
[ 'date:', '/', '/', '' ]
> text.split(/\D/);
[ '', '27', '01', '2007' ]
```

Objects

Basics

- objects are references to finite maps from **property keys** to values
- **property keys** can be arbitrary strings or **symbols**
- two basic ways to create objects
 - with an object literal
 - with **new** and a constructor

Examples

```
> let o1={}; // object literal
undefined
> let o2=new Object(); // equivalent definition with constructor Object
undefined
> let movie = {title:'2001: A Space Odyssey',
                 director:{firstname:'Stanley',surname:'Kubrick'},
                 year:1968};
undefined
> movie;
{ title: '2001: A Space Odyssey',
  director: { firstname: 'Stanley', surname: 'Kubrick' },
  year: 1968 }
```

Objects and properties

Basic rules

- properties' values can be read
- properties' values can be updated (with some exceptions)
- properties can be added/removed dynamically (with some exceptions)
- accessing an undefined property of an object returns **undefined**
- two basic ways for accessing/updating properties
 - usual **dot notation**, when property names are valid identifiers
 - more flexible **square brackets** notation
- accessing properties of **undefined** or **null** throws a `TypeError` exception

Objects and properties

Examples

```
> movie.title; // dot notation accepts only valid identifiers!  
'2001: A Space Odyssey'  
> movie['ti'+'tle']; // square brackets accept any expression  
'2001: A Space Odyssey'  
> movie.language='English'; // new property added  
'English'  
> movie.language;  
'English'  
> movie.language='Italian'; // property's value updated  
> movie.language;  
'Italian'
```

Objects and properties

Examples

```
> movie['running time']=161; // new property added
161
> movie['running time']; // dot notation ok only for valid identifiers
161
> movie.distribution;
undefined
> delete movie['running time']; // property removed
true
> movie;
{ title: '2001: A Space Odyssey',
  director: { firstname: 'Stanley', surname: 'Kubrick' },
  year: 1968,
  language: 'English' }
```

Objects and properties

Objects are manipulated by reference

```
> let movie = {title:'2001: A Space Odyssey',
               director:{firstname:'Stanley', surname:'Kubrick'},
               year:1968}
undefined
> let movie2 = movie; // copy by reference!
undefined
> movie2;
{ title: '2001: A Space Odyssey',
  director: { firstname: 'Stanley', surname: 'Kubrick' },
  year: 1968 }
> movie.language='English';
'English'
> movie2;
{ title: '2001: A Space Odyssey',
  director: { firstname: 'Stanley', surname: 'Kubrick' },
  year: 1968,
  language: 'English' }
```

Extended object literal syntax (since ES6)

Shorthand properties

```
> let x=1, y=2;  
undefined  
  
// before ES6  
> let o={x:x,y:y};  
undefined  
> o;  
{ x: 1, y: 2 }  
  
// since ES6  
> let o={x,y};  
undefined  
> o;  
{ x: 1, y: 2 }
```

Extended object literal syntax (since ES6)

Computed property names

```
> let name='p';
undefined

// before ES6
> let o={};
undefined
> o[name+1]=1; o[name+2]=2;
2
> o;
{ p1: 1, p2: 2 }

// since ES6
> let o={[name+1]:1,[name+2]:2};
undefined
> o;
{ p1: 1, p2: 2 }
```

JavaScript Object Notation (JSON)

In a nutshell

- a practical text-based data exchange open-standard format
- syntax very similar to JavaScript object literals
- property names must be **always wrapped in double quotes**

Example

```
> JSON.stringify({title:'Blade Runner',year:1982}); // returns a string
'{"title":"Blade Runner","year":1982}'  
  
> JSON.parse('{"firstname":"Stanley","surname":"Kubrick"}'); // returns an object
{ firstname: 'Stanley', surname: 'Kubrick' }  
  
> let m={title:'Blade Runner',
            director:{firstname:'Stanley',surname:'Kubrick' }}
undefined  
> JSON.stringify(m);
'{"title":"Blade Runner","director":{"firstname":"Stanley","surname":"Kubrick"}}'
```

JavaScript Object Notation (JSON)

Importing a JSON file

data in JSON format can be easily imported and converted into values

```
> const data = require('./data.json'); // reads and decodes data from a JSON file
```

JSON encoding of primitive values

```
> JSON.stringify(undefined);
undefined
> JSON.stringify(Symbol());
undefined
> JSON.stringify(null);
'null'
> JSON.stringify(42);
'42'
> JSON.stringify('hello');
'"hello"'
> JSON.stringify("hello");
'"hello"'
> JSON.stringify(true);
'true'
```

JavaScript Object Notation (JSON)

JSON encoding of BigInt is not allowed

```
> JSON.stringify(BigInt(42));  
Uncaught TypeError: Do not know how to serialize a BigInt  
    at JSON.stringify (<anonymous>)
```

JSON encoding of functions is undefined

```
> JSON.stringify(function() {});  
undefined
```

Properties associated with undefined/symbols/functions are not stringified

Unless stringify is customized...

```
> JSON.stringify({x:undefined, y:Symbol(), z:42});  
'{"z":42}'
```

Remark: inherited or not enumerable properties are **not** stringified (see later)

JavaScript Object Notation (JSON)

stringify can be easily customized

- by defining property `BigInt.prototype.toJSON`
- by defining property `Function.prototype.toJSON`
- by using `JSON.stringify(value, replacer)`

JavaScript Object Notation (JSON)

Examples

```
> let b=BigInt(30);
undefined
> JSON.stringify(b);
Uncaught TypeError: Do not know how to serialize a BigInt
    at JSON.stringify (<anonymous>)
> BigInt.prototype.toJSON=function(){return 'a big int';};
[Function (anonymous)]
> JSON.stringify(b);
'a big int'
> function replacer(key, value) {
  return (typeof value === 'symbol')? 'a symbol' : value;
}
undefined
> function myStringify(value) {
  return JSON.stringify(value, replacer);
}
undefined
> myStringify({ x: 1, y: Symbol() });
'{ "x":1,"y":"a symbol"}'
```

JSON schemas

An effective way to specify data formats and validate data

- a **JSON schema** defines the allowed shapes of JSON data
- several libraries can be used to validate data against a JSON schema

Example with the ajv library

```
const Ajv = require("ajv"); // imports the library
const ajv = new Ajv({ allErrors: true });
const schema = { // defines the schema
  type: "object",
  properties: {foo: { type: "integer" }, bar: { type: "string" }},
  required: ["foo"],
  additionalProperties: false
};
const validate = ajv.compile(schema);
// assuming data is already parsed from a JSON input
const data = {foo: 1, bar: "abc"};
if (!validate(data)) console.error(validate.errors);
```

Iterating over property keys

in operator

```
> let o={x:1,y:2};  
undefined  
> 'x' in o; // checks if 'x' is a property key of o, possibly inherited  
true  
> 'z' in o;  
false  
> o.z;  
undefined  
> o.z=undefined; // allowed, but not recommended  
undefined  
> 'z' in o;  
true
```

Iterating over property keys

for...in statement

```
// iterates over all enumerable string (inherited) property keys of o
> for(let k in o)console.log(k);
X
Y
Z
undefined
```

Arrays

In a nutshell

- exotic objects which are numerically indexed by default
- valid indexes: $+0 \leq i \leq 2^{53} - 1$
- an index is a canonical numeric String: a String representation of a Number
- two ways to create arrays
 - with an array literal
 - with **new** Array (...)

Example

```
> let a1=[1,'two',3]; // array literal
undefined
> let a2=new Array(1,'two',3);
undefined
> a1;
[ 1, 'two', 3 ]
> a2;
[ 1, 'two', 3 ]
```

Array length

All arrays have the `length` property

```
> a=[1,2,3];
[ 1, 2, 3 ]
> a.length;
3
```

The `length` property is writable!

```
> a=[1,2,3];
[ 1, 2, 3 ]
> a.length=1;
1
> a;
[ 1 ]
> a.length=3;
3
> a;
[ 1, <2 empty items> ]
```

Arrays are very flexible!

Arrays can be sparse

```
> let a=[];
undefined
> a[2]=2;
2
> a;
[ <2 empty items>, 2 ]
> a.length;
3
> 0 in a;
false
> a[0]=undefined;
undefined
> 0 in a;
true
> a;
[ undefined, <1 empty item>, 2 ]
> for(let i in a)console.log(i); // 'length' not enumerable
0
2
undefined
```

Arrays are very flexible!

Arrays are also objects!

```
> let a=['27']; a.index=5; a.input='date:27/01/2007';
'date:27/01/2007'
> a;
[ '27', index: 5, input: 'date:27/01/2007' ]
> a.length;
1
> typeof a;
'object'
> Array.isArray(a);
true
```

Arrays are not stringified as ordinary objects

```
> let a = [1,2,3]; JSON.stringify(a);
'[1,2,3]'
> a.key=5;
5
> JSON.stringify(a);
'[1,2,3]'
```

Remark: in arrays only indexed values are stringified; length, key are not enumerable

Some useful array functions

`push (e_0, \dots, e_n) / pop ()`

```
> let a=[1,2,3];
undefined
> a.push(0); // adds as last element and returns the new length of a
4
> a;
[ 1, 2, 3, 0 ]
> a.pop(); // removes the last element and returns it
0
> a;
[ 1, 2, 3 ]
```

`unshift (e_0, \dots, e_n) / shift ()`

```
> a.unshift(0); // adds as first element and returns the new length of a
4
> a;
[ 0, 1, 2, 3 ]
> a.shift(); // removes the first element and returns it
0
> a;
[ 1, 2, 3 ]
```

Some useful array functions

slice (*start, end*)

```
> a=[1,2,3,2];
[ 1, 2, 3, 2 ]
> a.slice(0,4); // returns a copy of a
[ 1, 2, 3, 2 ]
> a.slice(); // defaults are 0 and a.length
[ 1, 2, 3, 2 ]
> a.slice(1,3);
[ 2, 3 ]
```

join (*separator*)

```
> a='a,b,c'.split(',');
[ 'a', 'b', 'c' ]
> a.join(';');
'a;b;c'
> a.join(); // default separator is ','
'a,b,c'
```

Some useful array functions

`splice (start, deleteCount, item1, ..., itemn)`

```
> a=[1,2,3,2];
[ 1, 2, 3, 2 ]
> a.splice(0,2,4,5); // replaces the first two elements with 4 and 5
[ 1, 2 ]
> a;
[ 4, 5, 3, 2 ]
> a.splice(2,2); // removes the last two elements
[ 3, 2 ]
> a;
[ 4, 5 ]
```

Array functions based on functional programming patterns

sort (*compareFunction*)

```
> [4,3,2,1].sort(); // uses default order
[ 1, 2, 3, 4 ]
> ['ab','aa','a'].sort();
[ 'a', 'aa', 'ab' ]
> [121,25,4].sort(); // default order converts values into strings!!!
[ 121, 25, 4 ]
```

sort with anonymous user-defined sorting functions

```
> [121,25,4].sort(function(x,y){return x<y ? -1 : (x>y ? 1 : 0)});
[ 4, 25, 121 ]
> [1,2,3].sort(function(x,y){return x<y ? 1 : (x>y ? -1 : 0)});
[ 3, 2, 1 ]
```

Array functions based on functional programming patterns

forEach (*function*)

```
a = ['a', 'b', 'c'];
[ 'a', 'b', 'c' ]
> a.forEach(function(el, index, array) {console.log(el, index, array)});
a 0 [ 'a', 'b', 'c' ]
b 1 [ 'a', 'b', 'c' ]
c 2 [ 'a', 'b', 'c' ]
undefined
> a.forEach(function(el, index, array) {array[index]=el.toUpperCase();});
undefined
> a;
[ 'A', 'B', 'C' ]
```

Functions

In a nutshell

- functions are first-class, can be manipulated as ordinary values
- they are a specialized kind of objects, they can have properties
- introduced with **definition expressions** or **declaration statements**

Examples

```
function inc(x) {return x+1}; // declaration statement

let inc = function(x){return x+1}; // definition expression, anonymous

// recursive functions

// declaration statement
function fact(n){if(n<=1) return 1; else return n * fact(n-1)};

// definition expression, non anonymous
let fact=function f(n){if(n<=1) return 1; else return n * f(n-1)};
```

Function definition expressions

Remark

Each time the same function definition expression is evaluated, a **new** function object is created and returned

Same behavior as for object and array literals

Examples

```
> let f1=function(){}; let f2=f1; f1==f2;  
true  
> function gen_ob(){return {x:0,y:0}};  
undefined  
> function gen_ar(){return [1,2,3]};  
undefined  
> function gen_fu(){return function(){}};  
undefined  
> gen_ob()!==gen_ob() && gen_ar()!==gen_ar() && gen_fu()!==gen_fu();  
true
```

Function invocation

Four different kinds of invocations

- as functions
- as methods
- as constructors
- through `call()` and `apply()`

`this` keyword

as in many other object oriented languages, `this` has a special meaning if `this` is in the body of a function invoked as a simple function, then

- `this` is `undefined` in strict mode
- `this` is the global object in unrestricted mode (not recommended!)

```
> let strict=function(){return !this}();
```

Function calls

Arguments and parameters

- arguments passed by value
- num. of arguments does not need to equal num. of parameters

```
> function cat(x,y){return x+' '+y};  
undefined  
> let h='hello';  
undefined  
> let w='world';  
undefined  
> cat(h,w);  
'hello world'  
> cat(h);  
'hello undefined'  
> cat(h,w,'!');  
'hello world'
```

Parameter defaults (since ES6)

Rules

- **default values** can be defined for function parameters
- parameter default expressions are evaluated when the function is called
- parameter default expressions are evaluated only when the corresponding arguments are missing
- values of previous parameters can be used to define the default value of the parameters that follow them

Parameter defaults (since ES6)

Example

```
function getPropertyNames(o, a = []) {
  for(let prop in o) a.push(prop);
  return a;
}
function rectangle(width, height=width*2){
  return {width, height}; // ES6 shorthand for object literals
};
> getPropertyNames({x:1,y:1});
[ 'x', 'y' ]
> rectangle(1);
{ width: 1, height: 2 }
```

Rest parameters (since ES6)

Practical solution for defining **variadic functions** (variable number of parameters)

Rules

- the rest parameter must be the last formal parameter
- the rest parameter contains an array with the remaining arguments
- the rest parameter contains always a real array
- the rest parameter may not have a default initializer

Examples

```
function catAll(...args){ return args.join(' ') }

function max(first, ...others) {
  let res = first;
  others.forEach(function(el){if (el > res) res = el;})
  return res;
}
```

Predefined arguments object (older than ES6)

Less practical than rest parameters

- arguments: a pseudo-array, property of the function
- accessible also with the predefined local variable arguments

Example

```
> function f(){return f.arguments}; // arguments is a property of the function
undefined
> f(1,'a');
{ '0': 1, '1': 'a' } // a pseudo-array
> function g(){return arguments}; // arguments is a predefined local variable
undefined
> g(1,'a');
{ '0': 1, '1': 'a' }
> f.arguments;
null
> g.arguments;
null
> function h(){return arguments==h.arguments};
undefined
> h(1);
false // remark: arguments and h.arguments do not store the same object!
```

Predefined arguments object (older than ES6)

Examples

```
> function test_args(x,y) {return arguments[0]===x && arguments[1]===y};  
undefined  
> test_args(1,2);  
true  
> test_args();  
true  
> test_args(1,2,3);  
true  
> test_args({x:1,y:2}, {name:'temp',val:24.5});  
true
```

Predefined arguments object (older than ES6)

Examples

```
> function catAll1() {
  let res='';
  for(let i=0;i<arguments.length;i++)res+=arguments[i]+'\ '
  return res;
};

undefined
> function catAll2() {
  let res='';
  for(let i in arguments)res+=arguments[i]+'\ '
  return res;
}

undefined
> function catAll3(){ // recall: arguments is not an array!
  return Array.from(arguments).join(' ') // only since ECMAScript 6
};

undefined
> catAll1('a','b','c');
'a b c '
> catAll2('a','b','c');
'a b c '
> catAll3('a','b','c');
'a b c '
```

Defining and using function properties

A simple example

```
> function newInt(){return newInt.counter++};  
undefined  
> newInt.counter=0 ;  
0  
> newInt();  
0  
> newInt();  
1  
> newInt();  
2  
> newInt;  
{ [Function: newInt] counter: 3 }
```

Defining and using function properties

A more involved example: function+pseudo-array

```
> function fact(n) {
    if (Number.isInteger(n) && n>=0) {
        if (!(n in fact))
            fact[n] = n * fact(n-1);
        return fact[n];
    }
    else return NaN;
}
undefined
> fact[0] = 1;
1
> fact(5);
120
> fact;
{ [Function: fact] '0': 1, '1': 1, '2': 2, '3': 6, '4': 24, '5': 120 }
```

let declarations (since ES6)

Block scope supported by let

scope: the part of code affected by a variable declaration

```
> let x=0;  
undefined  
> for(let i=0;i<3;i++) {  
    let x=i;  
    console.log(x);  
}  
0  
1  
2  
undefined  
> x;  
0  
> i;  
Uncaught ReferenceError: i is not defined
```

var/function statements (older than ES6)

Less intuitive rules than let declarations

function scope supported, block scope not supported

local variables in functions hide global variables

```
> var level=0;  
undefined  
> function f() {  
    var level=1;  
    return level;  
};  
undefined  
> f();  
1  
> level;  
0
```

var/function statements (older than ES6)

Less intuitive rules than let declarations

function scope supported, block scope not supported

Block scope not supported by var/function statements

```
> var x=0;  
undefined  
> for(var i=0;i<3;i++) {  
    var x=i;  
    console.log(x);  
}  
0  
1  
2  
undefined  
> x;  
2  
> i;  
3
```

Variable hoisting with `let` declarations

Rule

in a block a variable cannot be accessed before its declaration, even when a variable with the same name is declared in an outer block

Example

```
let x=42;  
if(x>0){  
    console.log(x); // Error: Cannot access 'x' before initialization  
    let x=4;  
    console.log(x);  
};
```

Variable hoisting with `let` declarations

Rule

variables can be used in the right-hand-side of their declaration if they are not evaluated during their initialization

Example

```
> let f = function() {return f;}  
undefined  
> f() === f;  
true
```

Variable hoisting with `var` statements

Rules

- a variable declared within a function is visible throughout the whole body independently of the specific line where it has been declared
- a similar rule applies for variables declared at top level

Example

```
> var level=0;  
undefined  
> function f(){  
    console.log('level: '+level);  
    var level=1;  
    console.log('level: '+level);  
};  
undefined  
> f();  
level: undefined  
level: 1  
undefined
```

Variable hoisting with `function` statements

Rule

a function can be used in its scope before the line it is declared

Example

```
// script fun_hoist.js
'use strict';
console.log(dec(1));
function dec(x){return x-1;}; // fun declaration
console.log(inc(1));
function inc(x){return x+1;}; // fun declaration
```

#linux shell

```
$ node fun_hoist.js
0
2
```

Use of **var** (if needed ...)

Suggestions (if you really has to use **var**)

- **var** may be omitted for global variables **only** in unrestricted mode
- global variables declared with **var** **cannot** be deleted
- global variables declared **without var** can be deleted

Example 1

```
> x=0;  
0  
> var y=1;  
undefined  
> delete x;  
true  
> x;  
Thrown:  
ReferenceError: x is not defined  
> delete y;  
false  
> y;  
1
```

Use of **var** (if needed ...)

Suggestions (if you really has to use **var**)

- for declaring local variables in functions **var** is **always** required

Example 2

```
> function f() {
  x=1; // not a local declaration!
};
undefined
> f(); // strict mode
Uncaught ReferenceError: x is not defined
> f(); // unrestricted mode
undefined
> x; // in unrestricted mode 'x' declared as global variable!
1
```

Lexical scope

Rule

JavaScript uses **static (a.k.a. lexical) scope** for non local variables in functions

Example

```
> let res=42;
undefined
> function dec() {
    return res-1;
}
undefined
> function apply(f) {
    let res=1;
    return f();
}
undefined
> apply(dec);
```

Lexical scope

Rule

JavaScript uses **static scope** for non local variables in functions

Example

```
> let res=42;
undefined
> function dec() {
    return res-1;
}
undefined
> function apply(f) {
    let res=1;
    return f();
}
undefined
> apply(dec);
41
```

Global variables as properties

Rules

- global variables are properties of the **global object**
- the global object is referenced with
 - **this** (both server (Node.js)/client sides (browser))
 - or the self-referential property **global** (server-side only)
 - or the self-referential property **window** (client-side only)

Example (server-side with Node.js)

```
> let x=1;
undefined
> this.x;
1
> this.x++;
1
> x;
2
> this.x==global.x;
true
> this.global==global;
true
```

Methods

Definition

A **method** is a property key associated with a function

Method invocation

```
> let cell={value:0, set:function(v){this.value=v} };  
undefined  
> cell.set(42);    // select 'set' and call it  
undefined  
> cell.value;  
42  
> cell['set'](0); // more flexible syntax with square brackets  
undefined  
> cell.value;  
0
```

Invocation context of a method

When a function is invoked as a method, **this** denotes the object through which the method has been invoked

Methods

Shorthand methods without or with square brackets (since ES6)

```
> let cell1={value:0, set(v) {this.value=v} };  
undefined  
> const prefix='set';  
undefined  
> let cell2={value:0, [prefix+'Cell'](v) {this.value=v} };  
undefined  
> cell2;  
{ value: 0, setCell: [Function: setCell] }  
> const opaque=Symbol(); // symbolic property name  
undefined  
> let cell3={value:0, [opaque](v) {this.value=v} };  
undefined  
> cell3;  
{ value: 0, [Symbol()]: [Function (anonymous)] }  
> cell3[opaque](2);  
undefined  
> cell3.value;  
2
```

Property keys can be of type symbol (since ES6)

What are symbols?

symbols are opaque values

```
> let s1=Symbol(), s2=Symbol();
undefined
> s1;
Symbol()
> s2;
Symbol()
> s1==s2;
false
```

Property keys can be of type symbol (since ES6)

motivations for using symbols for property keys:

- implementation details can be hidden
- it offers a safe extension mechanism (see iterators later on)

Example

```
> function addProperty(obj, val){ // extends obj with no conflicts
  let s=Symbol();
  obj[s]=val; // adds the pair symbol-key s and value val
  return s; // returns the symbol for easier access
};

undefined
> let o={x:1,y:2}, newProp=addProperty(o, 42); // saves the new key
undefined
> o[newProp];
42
> Object.getOwnPropertySymbols(o); // array of non-inherited symbol keys
[ Symbol() ]
> Object.getOwnPropertySymbols(o)[0]==newProp;
true
> Object.getOwnPropertyNames(o); // array of non-inherited string keys
[ 'x', 'y' ]
> o.newProp; // no string-key 'newProp'
undefined
```

The bind () method

Example on the usefulness of bind ()

```
> let obj={getSelf(){return this;}};  
undefined  
> obj.getSelf() === obj; // method invocation  
true  
> let f = obj.getSelf;  
undefined  
> f==obj.getSelf;  
true  
> f() === obj; // function invocation  
false  
> f(); // f is unbound from 'obj', 'this' undefined in strict mode  
undefined
```

Remark

f () returns **undefined** in **strict mode** and the global object in **unrestricted mode** (not recommended)

The bind() method

bind returns a new function where **this** is associated with the passed argument

```
> let getObj=f.bind(obj); // binds 'this' in 'f' to 'obj'  
undefined  
> getObj() === obj;  
true
```

The bind() method

Another example

```
> let cell={value:0, set(v) {this.value=v} };  
undefined  
> let set=cell.set;  
undefined  
> let cell2 = {value:0};           // a new cell with no method  
undefined  
> let setCell2=set.bind(cell2);   // an external method of cell2  
undefined  
> setCell2(42);  
undefined  
> cell2;  
{ value: 42 }  
> cell;  
{ value: 0, set: [Function: set] }
```

The call () and apply () methods

useful when one does not want to permanently bind **this** in a function

Behavior

- $f.\text{call}(o, a_1, \dots, a_n)$: f is called on o with arguments a_1, \dots, a_n (variadic function)
- $f.\text{apply}(o, [a_1, \dots, a_n])$: arguments can be passed through an array or a pseudo-array

The call () and apply () methods

Example

```
> let cell={value:0, set(v) {this.value=v} };  
undefined  
> let set=cell.set;  
undefined  
> let cell2 = {value:0};  
undefined  
> set.call(cell2,42);      // binds this to cell2 in set  
undefined  
> cell2;  
{ value: 42 }  
> set.apply(cell2,[43]);   // binds this to cell2 in set  
undefined  
> cell2;  
{ value: 43 }  
> cell2.set;              // no internal method set  
undefined
```

The call () and apply () methods

Example of “monkey-patching” for logging method calls (meta-programming)

```
> function log(o, m) { // o an object, m the key of a method of o
    let original = o[m]; // saves the original method to be logged
    o[m] = function (...args) {
        console.log("Entering:", m);
        // 'this' is the object on which m will be called
        let result = original.apply(this, args);
        console.log("Exiting:", m);
        return result;
    };
}
undefined
> let cell = { value: -42, getAbs() { return Math.abs(this.value); } };
undefined
> log(cell, 'getAbs');
undefined
> console.log(cell.getAbs());
Entering: getAbs // printed by the patched method
Exiting: getAbs // printed by the patched method
42              // printed by console.log(cell.getAbs())
undefined
```

Constructors

Functions can also work as constructors

```
// convention: constructors' names start with a capital letter
> function Cell(v){this.value=v}; // adds & initializes a new property
undefined
> let cell=new Cell(42);
undefined
> cell;
Cell { value: 42 }; // note: the object "has dynamic type" Cell
```

Rules

- invocation through **new**
- a new object *o* is created, and the specified function is invoked with its arguments and **this** associated with *o*

Returned values

Rule for function invocation

- if no explicit value is returned, then `undefined` is returned

Rules for constructor invocation

- functions invoked as constructors usually do not use `return` statements
- their main purpose is object initialization
- in case of no `return` or `return e`, with `e` primitive, then `this` is returned
- but still constructors can return objects different from `this` ...

Returned values

Example

```
> function Num(n) { // works in strict mode
    if (this) // here equivalent to this!=undefined
        this.value = n;
    return Number(n);
}
undefined
> new Num(42); // called as constructor
Num { value: 42 }
> Num(42); // called as a conversion function
42
> Num ('0xf');
15
```

Equalities

Three kinds of equalities

- strict equality ===
- Object.is
- loose equality ==

==== and Object.is

- intuitive behavior, they can hold only when values have the same type, no implicit conversions
- their behavior slightly differs on numbers

```
> -0 === 0;  
true  
> Object.is(-0, 0);  
false  
> NaN === NaN;  
false  
> Object.is(NaN, NaN);  
true
```

Loose equality

Remark

Not intuitive behavior, hard to be predicted in some cases when values are involved
Operands can be implicitly converted

Examples

```
> null == undefined;  
true  
> '0' == 0;  
true  
> 0 == false;  
true  
> '0' == false;  
true  
> [1] == 1;  
true  
> 1 == new Number(1);  
true  
> [1] == new Number(1);  
false
```

Accessor properties (a.k.a. getters/setters)

Example

```
> let o={v:1, get temperature(){return this.v+' Celsius';},  
         set temperature(i){if(Number.isInteger(i)) this.v=i;}};  
undefined  
> o.temperature=3; // set temperature is called with argument 3  
3  
> o.temperature;    // get temperature is called  
'3 Celsius'  
> o;  
{ v: 3, temperature: [Getter/Setter] }  
> o.temperature='4';  
'4'  
> o;  
{ v: 3, temperature: [Getter/Setter] }
```

- temperature is called an **accessor** property
- non-accessor properties are simply called **data properties**

Accessor properties (a.k.a. getters/setters)

Remarks

- accessor properties are getter/setter methods (as C# properties)
 - useful to have **more controlled access** on data in objects
- values of data properties can be also functions (that is, methods)
- behavior of `JSON.stringify`: setters are ignored, getters are executed and then their returned values are stringified

```
> JSON.stringify(o);
'{"v":3,"temperature":"3 Celsius"}'
```

Remark: throughout this course we mainly focus on data properties

Property information retrieval

There are several ways to retrieve information on objects' properties

```
> let a=[1,2];
undefined
// tests whether 'length' is (a possibly inherited) key of 'a'
> 'length' in a;
true
// loops over the (possibly inherited) enumerable string-keys of 'a'
> for(const i in a)console.log(i);
0
1
undefined
// returns an array of the key-value pairs for all non-inherited enumerable
// string-keys of 'a'
> Object.entries(a);
[ [ '0', 1 ], [ '1', 2 ] ]
// returns an array of all non-inherited string-keys of 'a'
> Object.getOwnPropertyNames(a);
[ '0', '1', 'length' ]
```

Property information retrieval

There are several ways to retrieve information on objects' properties

```
// returns an array of all non-inherited symbol-keys of 'a'  
> a[Symbol()]=2;  
2  
> Object.getOwnPropertySymbols(a)  
[ Symbol() ]  
// returns true iff 'length' is a non-inherited key of 'a'  
> Object.hasOwnProperty(a,'length');  
true  
// returns an array of all non-inherited enumerable string-keys of 'a'  
> Object.keys(a);  
[ '0', '1' ]  
// returns an array of the values for all non-inherited enumerable string-keys of  
// 'a'  
> Object.values(a);  
[ 1, 2 ]
```

Property information retrieval

Table summarizing all operators for retrieving properties

	inherited	non-enum	string-key	symbol-key
in	yes	yes	yes	yes
for-in	yes	no	yes	no
entries	no	no	yes	no
getOwnPropertyNames	no	yes	yes	no
getOwnPropertySymbols	no	yes	no	yes
hasOwn	no	yes	yes	yes
keys	no	no	yes	no
values	no	no	yes	no

Property descriptors

Properties have **attributes** specified through descriptor objects

Example with data properties

```
> let a=['a','b'];
undefined
// returns all own property descriptors of 'a'
> Object.getOwnPropertyDescriptors(a);
{
  '0': { value: 'a', writable: true, enumerable: true, configurable: true },
  '1': { value: 'b', writable: true, enumerable: true, configurable: true },
  length: { value: 2, writable: true, enumerable: false, configurable: false }
}
```

Property descriptors

Properties have **attributes** specified through descriptor objects

Example with accessor properties

```
> let o={v:1, get temperature(){return this.v+' Celsius';},
         set temperature(i){if(Number.isInteger(i)) this.v=i;}};
undefined
// returns all own property descriptors of 'o'
> Object.getOwnPropertyDescriptors(o);
{
  v: { value: 1, writable: true, enumerable: true, configurable: true },
  temperature: {
    get: [Function: get temperature],
    set: [Function: set temperature],
    enumerable: true,
    configurable: true
  }
}
```

Attributes of data properties

Four attributes

- **value**: the value associated with the data property
- **writable**: is it possible to update its value?
- **enumerable**: is it visible to **for in** statement or `Object.keys()` ?
- **configurable**: can we delete it, or arbitrarily change its attributes?

Selected rules for data properties

if a property is **not configurable**, then:

- its **configurable** or **enumerable** attributes **cannot** be changed
- for the **writable** attribute **true** → **false** allowed, **false** → **true** **not** allowed
- its **value** attribute **cannot** be changed if it is also **not writable**
the **value** attribute of a property can be changed with `Object.defineProperty`
if it is **configurable** (**non-writable** means **non updatable** through assignment)

Remark: rules are simplified, accessor properties not considered

Definition of property attributes

Example 1

```
> let p={x:1};  
undefined  
> Object.getOwnPropertyDescriptor(p,'x');  
{ value: 1, writable: true, enumerable: true, configurable: true }  
> Object.defineProperty(p,'y',{value:2,writable:true});  
{ x: 1 }  
> Object.getOwnPropertyDescriptor(p,'y');  
{ value: 2, writable: true, enumerable: false, configurable: false }  
> p.y;  
2  
> p.y=3;  
3  
> p.y;  
3  
> Object.defineProperty(p,'y',{writable:false});  
{ x: 1 }  
> p.y=4; // throws TypeError in strict mode
```

Definition of property attributes

Example 2

```
> let p={};  
undefined  
> Object.defineProperty(p, 'x', {value:1,configurable:true});  
{ }  
> Object.getOwnPropertyDescriptor(p,'x');  
{ value: 1, writable: false, enumerable: false, configurable: true }  
> p.x=2; // throws TypeError in strict mode  
> Object.defineProperty(p,'x',{value:2});  
{ }  
> p.x;  
2
```

Iterable objects (since ES6)

In a nutshell

- objects where it is possible to loop over their data with **for-of** statement
- typical examples: arrays, strings, Set and Map objects, . . .

Example

```
> function sumAll(iter) { // iter should be an iterable object
    let sum = 0;
    for(const i of iter) sum += i;
    return sum;
};

undefined
> sumAll([1, 2, 3]);
6
> sumAll(new Set().add(1).add(2).add(3)); // sums all els in {1,2,3}
6
```

Iterable objects (since ES6)

Spread operator

Iterators can be used with the **spread operator** ... to expand an iterable object into an array or object literal or function invocation:

```
> [...'abcd'] // strings are iterable
[ 'a', 'b', 'c', 'd' ]
> {...'abcd'} // returns a pseudo array
{ '0': 'a', '1': 'b', '2': 'c' }
> let data = [1, 2, 3, 4, 5];
undefined
> Math.max(...data); // Math.max is a variadic function
5
> Math.max(data); // the maximum of an array is undefined
NaN
```

Iterator objects (since ES6)

Iterable and iterator objects

- an iterable object is any object with a special **iterator method** that returns a new object able to **iterate** over the elements of an object
- the key of the **iterator method** is always the symbol defined by `Symbol.iterator` (to ensure backward compatibility)
- iterators have the `next()` method which returns an **iteration result**
- an **iteration result** is an object with properties `value` and `done`

Iterator objects (since ES6)

Examples

```
> let it = [1,2,3][Symbol.iterator](); // returns a new iterator of the array
undefined
> it.next();
{ value: 1, done: false }
> let iterable = [1,2,3], iterator = iterable[Symbol.iterator]();
undefined
> for(let result = iterator.next(); !result.done; result = iterator.next())
  {console.log(result.value);}
1
2
3
// a more compact syntax
> for(const v of iterable)
  {console.log(v);}
1
2
3
undefined
```

Iterator objects (since ES6)

Iterator objects are themselves iterable

- the method `Symbol.iterator` of an iterator just returns itself
- occasionally useful for iterating through a “partially used” iterator

Example

```
> let str = 'abcd', iter = str[Symbol.iterator](), first = iter.next().value;  
undefined  
> first;  
'a'  
> let rest = [...iter];  
undefined  
> rest;  
[ 'b', 'c', 'd' ]
```

Destructuring assignment

Examples

Iterables can be used with **destructuring assignment**:

```
> let [x,y]='abc';
undefined
> x;
'a'
> y;
'b'
```

Destructuring assignment works also with object literals (independently of iterators):

```
> let o={r:1,q:2,p:3};
undefined
> let {p,q}=o;
undefined
> p;
3
> q;
2
```

Destructuring assignment

Examples

Destructuring assignment works also with iterables in conjunction with object literals:

```
> let a = [{x1:1,y1:2},{x2:3,y2:4}];  
undefined  
> let [{x1,y1},{x2,y2}]=a;  
undefined  
> x1;  
1  
> y1;  
2  
> x2;  
3  
> y2;  
4
```

Generator functions and iterators

- the **function*** declaration creates a binding of a new generator function to a given name. A generator function can be exited and later re-entered, with variable bindings saved across re-entrances
- each time a generator function is called, it returns a new Generator object, which behaves like an iterator
- the **yield** operator is used inside a generator function body to pause and resume it

Example

```
function* range(min, max) {
  for (let i = min; i < max; i++)
    yield i;
}

for (const i of range(2, 4))
  console.log(i); // prints 2 and 3
```

Generator functions

- a generator can exit and terminate the iteration with the **return** statement
- if a value is returned, it will be set as the value property of the object returned by the generator

Example

```
function* range(min, max) {
  for (let i = min; i < max; i++)
    yield i;
  return 'done';
}
let it = range(2, 4)[Symbol.iterator]();
let res = it.next();
for (; !res.done; res = it.next()) { console.log(res); }
console.log(res); // prints { value: 'done', done: true }
```

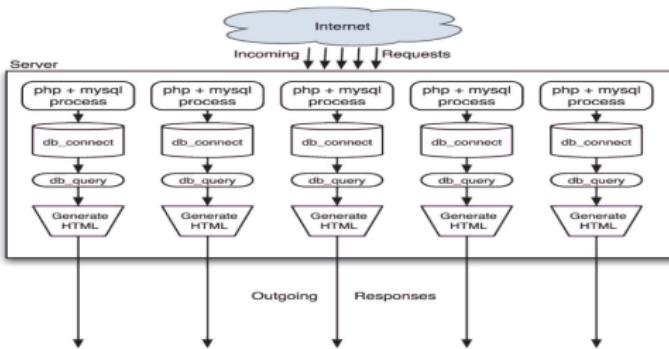
What is Node.js?

Brief history and motivation

- a platform for server-side applications based on JavaScript
- development started by Ryan Dahl at Joyent in 2009
- built on top of Chrome V8 and the libuv library for asynchronous I/O with event loops
- based on an **event-driven nonblocking I/O model**
- easy support for programming lightweight servers able to manage a considerable number of simultaneous requests

Why Node.js?

The traditional way to handle I/O



- I/O: databases connection, server communication, fs access, ...
- **drawbacks**
 - efficient task scheduling in concurrent programs is challenging
 - overhead: wasted memory and time
 - most of the time is spent for blocked computations: wasted CPU power

Why Node.js?

A simple example of synchronous (blocking) I/O

```
const {openSync,readSync}=require('fs') // uses 'openSync', 'readSync' of 'fs'  
const BUFFER_SIZE=2**20;  
  
// blocking I/O operation: the program waits until the file is open  
let fd=openSync(process.argv[2],'r');  
  
const buff=Buffer.allocUnsafe(BUFFER_SIZE);  
  
// blocking I/O operation: the program waits until the file is read  
// puts BUFFER_SIZE bytes of read data in buff at position 0 and advances  
let bytesRead=readSync(fd,buff);  
  
console.log(buff.toString('utf8',0,bytesRead));  
console.log(`File ${process.argv[2]} successfully read`);
```

Why Node.js?

Some technical details

- global function `require`: used to import CommonJS modules and JSON files
- global variable `process`
 - provides useful functionalities and information on the runtime environment
 - example: `process.argv` is the string array of command-line arguments
 - `process.argv[0]` contains the path to the node interpreter
 - `process.argv[1]` contains the path to the executed script, if any
- ES6 template strings
 - `'ERROR: ${err.code} (${err.message})'` is equivalent to
`'ERROR: '+err.code+' ('+err.message+')'`
 - expressions delimited by `${}` are evaluated, converted to strings, and concatenated

The Node.js style

Asynchronous functions

Behavior of an asynchronous call

- the aim of a call to an asynchronous function is to require some **I/O operation**
- the call **immediately returns** and **does not wait** for the completion of the I/O operation
- all the code following the call is executed **without waiting** for the completion of the I/O operation

The Node.js style

Callback functions

Handling of the completed I/O operation

- when the execution of all code after the call to the asynchronous function eventually terminates and the required I/O operation is completed, new code can be run
- the new code is specified by the asynchronous call by providing a **callback function**
- the **last parameter** of an asynchronous function is always the associated callback function

Parameters of the callback function

- the **first parameter** is used to notify **errors**: **null** means that the operation completed successfully, otherwise an error (= exception) is passed to inform on the reasons of the failure
- the other parameters depend on the specific I/O operation and provide the necessary information associated with the successful completion of the operation

The Node.js style

An example of asynchronous non blocking I/O

```
const { open, read } = require('fs');
const BUFFER_SIZE = 2 ** 20;
function handleError(err) {
    console.error(`ERROR: ${err.code} (${err.message})`);
}
open(process.argv[2], 'r',
    function (err, fd) { // callback called by open
        if (err) return handleError(err);
        const buff = Buffer.allocUnsafe(BUFFER_SIZE);
        read(fd, buff,
            function (err, bytesRead, buffer) { // callback called by read
                if (err) return handleError(err);
                console.log(buffer.toString('utf8', 0, bytesRead));
            });
    });
console.log(`Still reading file ${process.argv[2]}`);
```

The Node.js style

Callbacks and continuation passing style (CPS)

- callbacks specify how the computation has to continue after the requested operation has completed
- callback for open:

```
function (err, fd) { ... }
```

After the open operation completes the callback is called by the system

- with err bound to **null**, and fd to the corresponding file descriptor, if the operation has succeeded
- with err bound to an Error object, if the operation has failed

- callback for read:

```
function (err, bytesRead, buf) { ... }
```

After the read operation completes the callback is called by the system

- with err bound to **null**, bytesRead to the number of read bytes, and buf to the used buffer, if the operation has succeeded
- with err storing an Error object, if the operation has failed

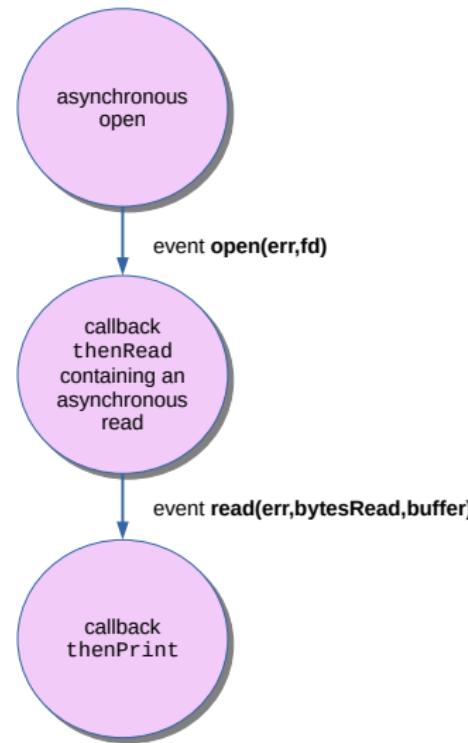
The Node.js style

An equivalent version with non-anonymous, non-nested callbacks

```
const { open, read } = require('fs');
const BUFFER_SIZE = 2 ** 20;
function handleError(err) {
    console.error(`ERROR: ${err.code} (${err.message})`);
}
function thenRead(err, fd) { // callback called by open
    if (err) return handleError(err);
    read(fd, Buffer.allocUnsafe(BUFFER_SIZE), thenPrint);
}
function thenPrint(err, bytesRead, buffer) { // callback called by read
    if (err) return handleError(err);
    console.log(buffer.toString('utf8', 0, bytesRead));
}
open(process.argv[2], 'r', thenRead);
console.log(`Still reading file ${process.argv[2]}`);
```

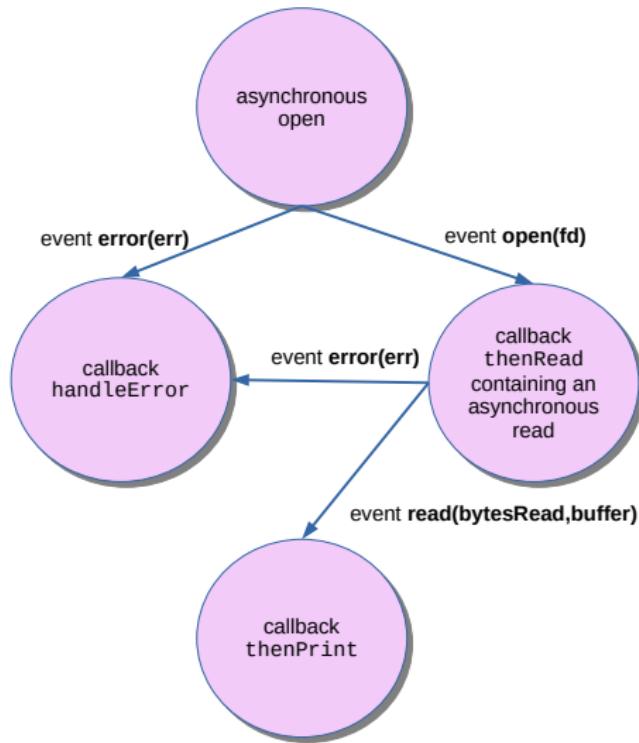
The Node.js style

Pictorial view of the control flow



The Node.js style

More detailed pictorial view of the control flow



Nested functions and closures

Two JavaScript notions essential for Node.js

- nested functions
- closures

Example to recall scope rules

```
> function outerLevel() {  
    let level=1;  
    function nestedLevel() {  
        let level=2;  
        return level;  
    }  
    return nestedLevel();  
}  
undefined  
> outerLevel();  
2
```

this, arguments and nested functions

Example

```
> let o = {
  test() {
    let self = this;
    console.log(this === o);
    f();

    function f() {           // f() has its own 'this' and 'arguments'
      console.log(this === o); // false in strict/unrestricted mode
      console.log(self === o);
    }
  }
}
> o.test();
true
false
true
undefined
```

this, arguments and arrow functions

Example with ES6 arrow function

```
> let o = {
  test() {
    console.log(this === o);
    let f = () => console.log(this === o); // no 'this'/'arguments'
    f();
  }
}
> o.test();
true
true
undefined
```

this, arguments and arrow functions

ES6 arrow functions do not have **this** and arguments

Syntax:

```
// simple cases
arg => expr
(arg1, ..., argn) => expr
// more general case
(arg1, ..., argn) => {stmt1; ...; stmtk}
```

The Node.js style

Version with nested arrow functions

```
const { open, read } = require('fs');
const BUFFER_SIZE = 2 ** 20;
function handleError(err) {
    console.error(`ERROR: ${err.code} (${err.message})`);
}
open(process.argv[2], 'r',
    (err, fd) => { // callback called by open
        if (err) return handleError(err);
        const buff = Buffer.allocUnsafe(BUFFER_SIZE);
        read(fd, buff,
            (err, bytesRead, buffer) => { // callback called by read
                if (err) return handleError(err);
                console.log(buffer.toString('utf8', 0, bytesRead));
            });
    });
console.log(`Still reading file ${process.argv[2]}`);
```

Nested functions

Example to recall scope rules

```
> function outerLevel() {
    let level=1;
    function incLevel(){level++;}
    function nestedLevel(){
        let level=42;
        incLevel();
        return level;
    }
    return nestedLevel();
}
undefined
> outerLevel();
```

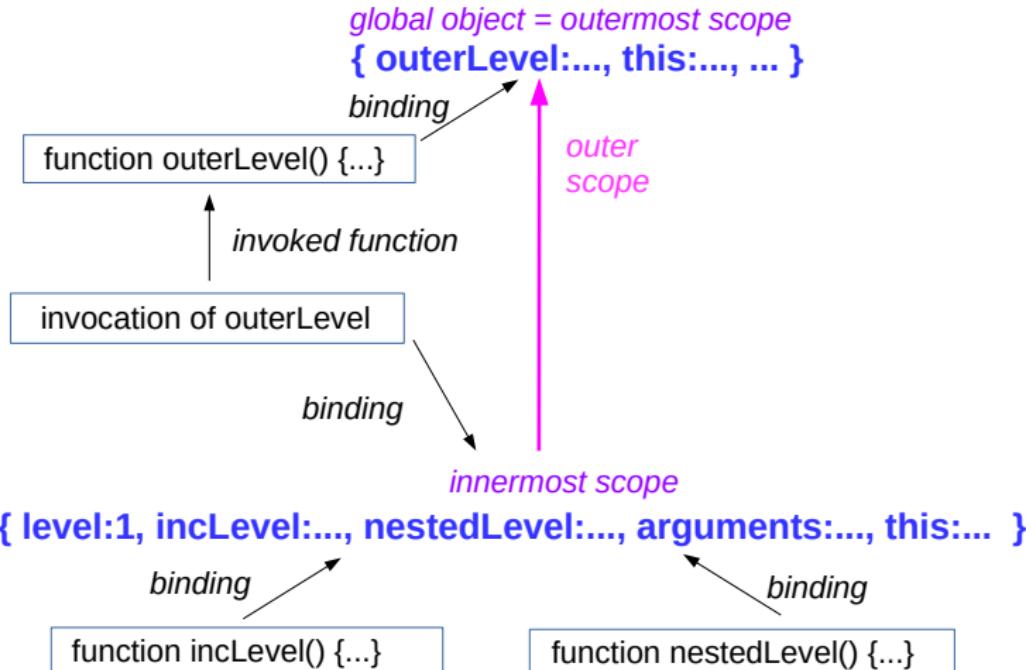
Nested functions

Example to recall scope rules

```
> function outerLevel() {
  let level=1;
  function incLevel(){level++;}
  function nestedLevel() {
    let level=42;
    incLevel();
    return level;
  };
  return nestedLevel();
}
undefined
> outerLevel();
42
```

The scope chain

Snapshot of the scope chain while `outerLevel` is running
 Situation just before execution of the last statement of `outerLevel`



The scope chain

Rules, part 1

- the **scope chain** is the list of objects corresponding to the nested scopes that define the variables that are in scope for a script/function body
- each time a function is created, the corresponding object refers to the first object in the chain (= immediately enclosing scope)
- each time a function is called, an object is created to hold the local variables for that call, and is added at the beginning of the scope chain
- **variable resolution:** a variable `x` is looked up starting from the first object in the chain (= innermost scope); if not found in the whole chain, then a `ReferenceError` exception is thrown

The scope chain

Rules, part2: external references to nested functions

- when a function call returns, the innermost scope is removed from the scope chain
- however, if a nested function f can be **referenced externally**, then the function object for f and its immediately enclosing scope are not reclaimed

Closures

Closure = function object + variable binding

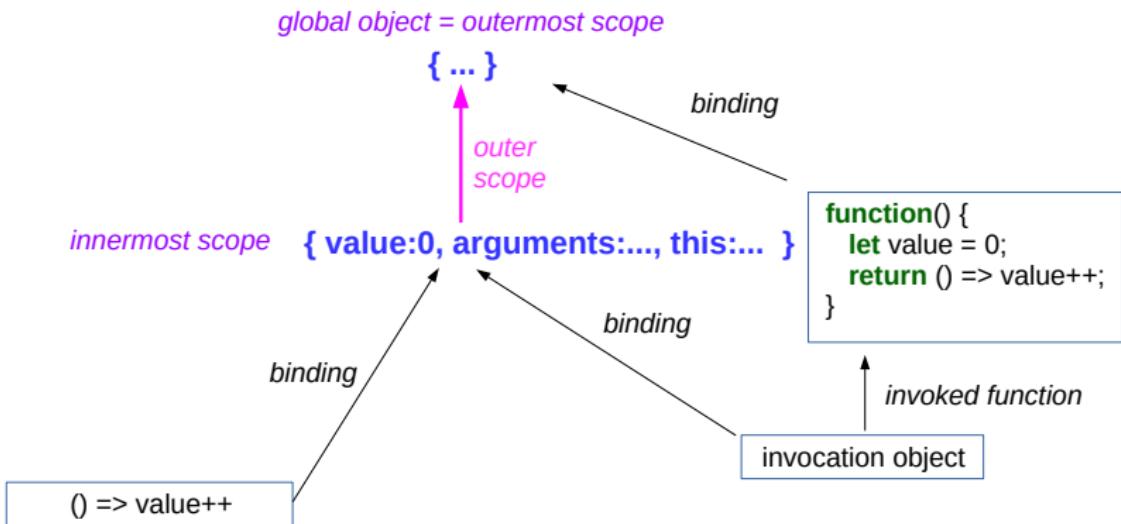
```
> let newInt = function () {
  let value = 0;
  return () => value++;
}(); // remark: the outer anonymous function is called with no args
undefined
> newInt
[Function]
> newInt()
0
> newInt()
1
```

Remarks

- arrow function `() => value++` is nested
- `() => value++` is returned by its outer anonymous function
- the outer function is called once and not referenced, `() => value++` is referenced externally through the global variable `newInt`

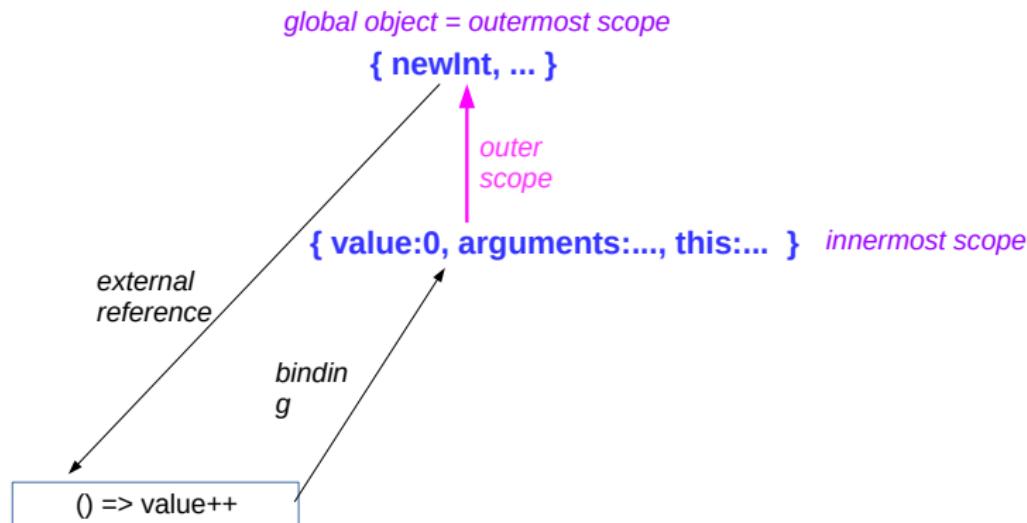
Pictorial view

Before completion of `return () => value++`



Pictorial view

After completion of `let newInt = function () { ... } ()`



Non-standard recursion in Node.js patterns

Nested functions, callbacks and scope rules allow non-standard forms of recursion

Example 1

```
const client = net.createConnection({ port: 8124 }, () => {
  console.log('connected to server!');
  client.write('world!\r\n');
});
```

The definition of `client` is correct because

- scope rules (same as for `let`) make `client` accessible in the arrow function
- the arrow function (=callback) is called after `net.createConnection` has returned the object reference associated with `client`

Remark: no recursive call is involved

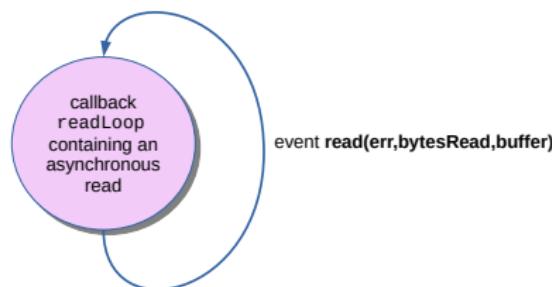
Non-standard recursion in Node.js patterns

Nested functions, callbacks and scope rules allow non-standard forms of recursion

Example 2

```
function readLoop(err,bytesRead,buffer) {  
  ...  
  if(bytesRead>0) read(fd,buffer,readLoop) ;  
  ...  
}
```

Remark: `readLoop` uses itself as callback of the asynchronous call of `read`, **no recursive call** is involved



Asynchronous functions and non-determinism

Order of I/O callback executions

it depends on

- the scheduling of I/O async operations
- the time required for completing the operations

Which file will be printed first?

```
const { open, read } = require('fs');
const BUFFER_SIZE = 2 ** 20;
function handleError(err) {
    console.error(`ERROR: ${err.code} (${err.message})`);
}
function thenRead(err, fd) { // callback called by open
    if (err) return handleError(err);
    read(fd, Buffer.allocUnsafe(BUFFER_SIZE), thenPrint);
}
function thenPrint(err, bytesRead, buffer) { // callback called by read
    if (err) return handleError(err);
    console.log(buffer.toString('utf8', 0, Math.min(6,bytesRead)));
}
open(process.argv[2], 'r',thenRead);
open(process.argv[3], 'r',thenRead);
```

Asynchronous functions and non-determinism

Order of I/O callback executions

it depends on

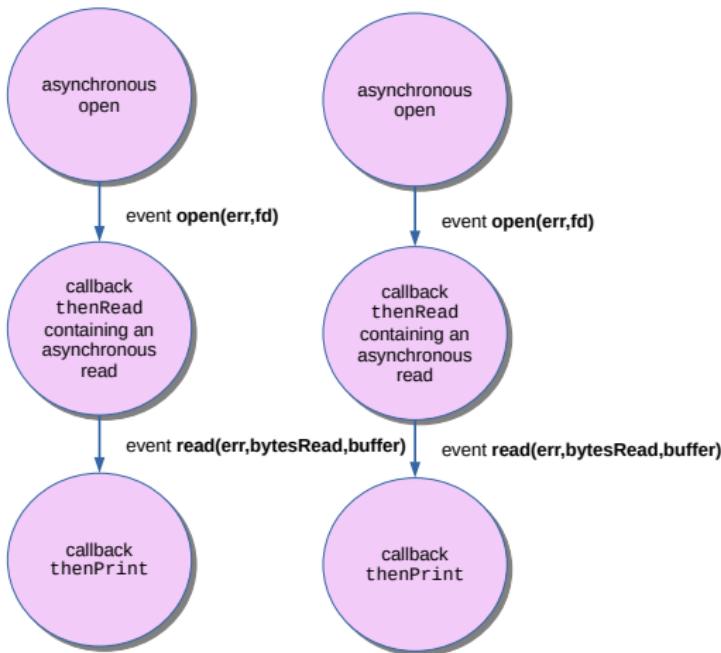
- the scheduling of I/O async operations
- the time required for completing the operations

Which file will be printed first? In this case the behavior is **non-deterministic** and partly depends on the dimensions of the files

```
const { open, read } = require('fs');
const BUFFER_SIZE = 2 ** 20;
function handleError(err) {
    console.error(`ERROR: ${err.code} (${err.message})`);
}
function thenRead(err, fd) { // callback called by open
    if (err) return handleError(err);
    read(fd, Buffer.allocUnsafe(BUFFER_SIZE), thenPrint);
}
function thenPrint(err, bytesRead, buffer) { // callback called by read
    if (err) return handleError(err);
    console.log(buffer.toString('utf8', 0, Math.min(6,bytesRead)));
}
open(process.argv[2], 'r',thenRead);
open(process.argv[3], 'r',thenRead);
```

Asynchronous functions and non-determinism

Example of non-deterministic behavior with parallel scheduling



The Node.js Event Loop

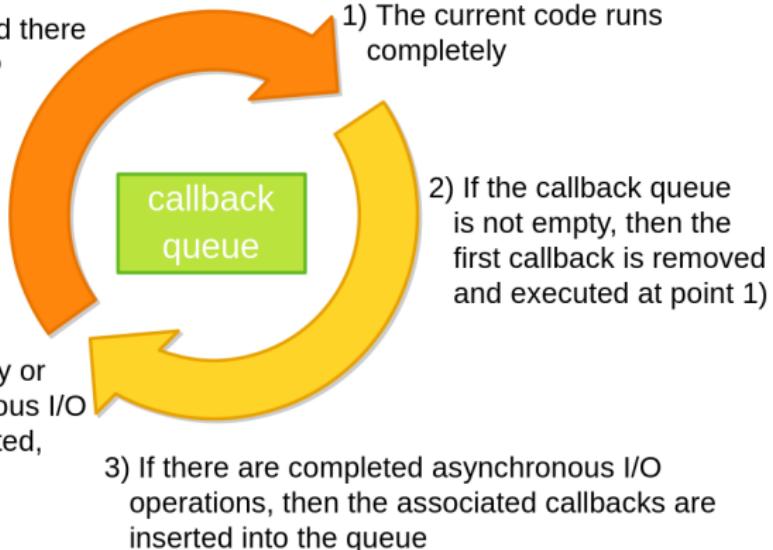
The event loop and the event/callback queues

- Node.js runs in a **single thread** in a **single process**
- it executes a **single event loop** at time
- calls to asynchronous functions **trigger events** associated with **callbacks**
- **pending events** are **enqueued** with their corresponding callbacks
- Node.js exits only if the currently running code has **finished executing** and there are **no pending events/callbacks**

The Node.js Event Loop

Simplified event loop in Node.js

5) If the queue is empty and there are no asynchronous I/O operations to complete, then the program terminates



The Node.js Event Loop

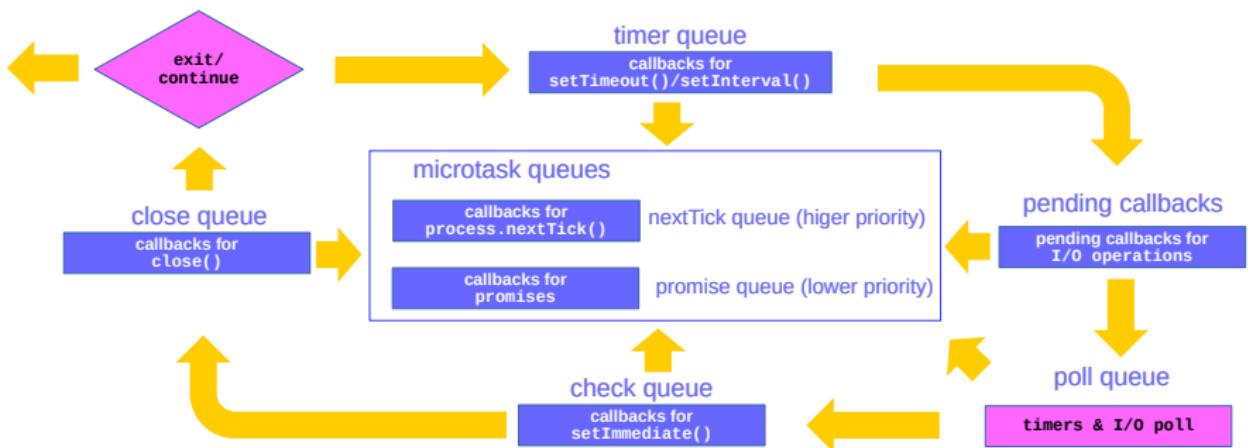
Remark: if the currently running code does not terminate, then there will be no next loop to process the pending events

Example of starvation

```
const { open, read } = require('fs');
const BUFFER_SIZE = 2 ** 20;
function handleError(err) {
    console.error(`ERROR: ${err.code} (${err.message})`);
}
open(process.argv[2], 'r',
  (err, fd) => {
    if (err) return handleError(err);
    const buff = Buffer.allocUnsafe(BUFFER_SIZE);
    read(fd, buff,
      (err, bytesRead, buffer) => { // callback called by read
        if (err) return handleError(err);
        console.log(buffer.toString('utf8', 0, bytesRead));
      });
  });
while (true) { // starvation!
  // does something without ever exiting the loop
}
```

The Node.js Event Loop

A more detailed overview (still with some simplifications)

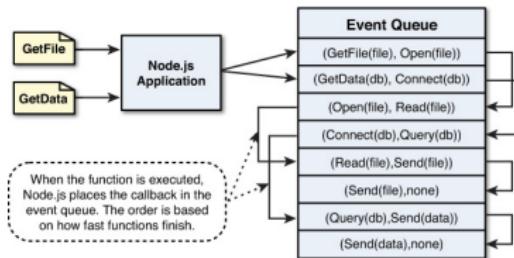


The Node.js Event Loop

Some details

- there are several **phases** and **more callback queues**:
 - **timers**: handles callbacks scheduled with `setTimeout()`, `setInterval()`
 - **I/O**: handles callbacks associated with asynchronous I/O operations
 - **check**: handles callbacks scheduled with `setImmediate()`
 - **close**: handles callbacks associated with asynchronous I/O operations to close resources
- the **poll phase** controls timed-out timers and completed I/O asynchronous operations
- callbacks in the queues are executed synchronously and in order
- the next step or callback is executed only when the execution of the current callback has terminated
- when the queue of a phase is empty or a limit is reached, the event loop moves to the next phase
- exception: the poll phase can wrap back to the timers phase if some timer timed-out and there are no callbacks scheduled with `setImmediate()`

The Node.js Event Loop



The poll phase

- new events processed in the poll phase are queued by the kernel
- events can be queued also during the poll phase
- iterates through its queue; if it is empty there are several options:
 - ➊ if there are ready timers callbacks, then the event loop wraps back to the timers phase
 - ➋ if there are callbacks scheduled by `setImmediate()`, then the event loop skips to the check phase
 - ➌ otherwise, the event loop waits for callbacks to be added to the queue and executes them immediately

The Node.js Event Loop

`process.nextTick()`

- a special asynchronous function
- callbacks are scheduled in `nextTickQueue`
- in any phase, callbacks in `nextTickQueue` are resolved before the event loop continues
- the callbacks always run after the rest of the user's code and before the event loop is allowed to proceed

Example

```
const { readFile } = require('fs');
readFile(process.argv[2],
  err => { if (!err) console.log('file operation'); }
);
setTimeout(console.log, 4, 'setTimeout');
setImmediate(console.log, 'setImmediate');
process.nextTick(console.log, 'nextTick');
console.log('main script');
```

The Node.js Event Loop

```
process.nextTick()
```

- a special asynchronous function
- callbacks are scheduled in `nextTickQueue`
- in any phase, callbacks in `nextTickQueue` are resolved before the event loop continues
- the callbacks always run after the rest of the user's code and before the event loop is allowed to proceed

Output

```
main script      // will be always printed on the 1st line
nextTick        // will be always printed on the 2nd line
setImmediate
setTimeout
file operation
```

Node.js: pros and cons

Advantages

- conciseness
 - event-driven programming simpler than concurrent programming
- Remark:** a thread pool is used under the hood, but in most cases concurrency is 'hidden'
- a huge library of modules
 - efficiency: lower resource requirements, operating system kernel features exploited as much as possible

Drawbacks

- asynchronous non blocking I/O with callbacks may be sometimes confusing
- development get more complex for servers requiring more computationally intensive activity

Managing errors with synchronous calls

A correct example

```
const {openSync}=require('fs');
try{
    openSync(process.argv[2], 'r');
    console.log(`File ${process.argv[2]} successfully opened`);
}
catch(err){
    console.error(`Error: ${err.message}`);
}
```

Managing errors with asynchronous calls

Example of incorrect code

```
const {open}=require('fs');
try{
  open(process.argv[2], 'r', (err, fd)=>{
    if(err) throw err;
    console.log(`File ${process.argv[2]} successfully opened`);
  })
}
catch(err){
  console.error(`Error: ${err.message}`);
}
```

Solutions

- pass error objects around as the first argument of callbacks
- better solutions: use **promises** or **async/await** (see next lectures)

The http module

A very simple HTTP client

```
const { request } = require('http');
const options = { port: 8080, path: '/${process.argv[2]} || ' };

const req = request(options, res => { // req=request, res=response
  console.log('HEADERS: ${JSON.stringify(res.headers)}');
  const chunks = [];
  res.setEncoding('utf8');
  res.on('data', chunk => chunks.push(chunk)); // collects chunks from the server
  res.on('end', () => console.log(chunks.join(''))); // gets the response
});

req.on('error', e => {
  console.error('problem with request: ${e.message}');
});

req.end(); // sends an empty GET request
```

The http module

A very simple HTTP client (equivalent definition)

```
const { request } = require('http');
const options = { port: 8080, path: `/${process.argv[2] || ''}` };

const req = request(options);

req.on('response', res => { // req=request, res=response
  console.log('HEADERS: ${JSON.stringify(res.headers)}');
  const chunks = [];
  res.setEncoding('utf8');
  res.on('data', chunk => chunks.push(chunk)); // collects chunks from the server
  res.on('end', () => console.log(chunks.join(''))); // gets the response
});

req.on('error', e => {
  console.error(`problem with request: ${e.message}`);
});

req.end(); // sends an empty GET request
```

The http module

A very simple HTTP server (example with GET requests)

```
const { readdir } = require('fs');
const { createServer } = require('http');
const home = '/home';
const headers = { "Content-Type": "application/json" };
const s = createServer(
  (req, res) => { // tacitly assuming that res.method==='GET'
    readdir(home + req.url,
      (err, files) => {
        if (err) {
          res.writeHead(500, headers);
          res.end(`${err.message}\n`);
        }
        else {
          res.writeHead(200, headers);
          res.end(` ${JSON.stringify(files)}\n`);
        }
      });
    console.log(`Request: ${req.method} URL: ${req.url}`);
  })
s.listen(8080); // use curl -iX GET localhost:8080/<path> or the Node.js client
```

The http module

A very simple HTTP server (example with POST requests)

```
const { createServer } = require('http');
const headers = { "Content-Type": "application/json" };
const s = createServer(
  (req, res) => {
    const chunks = [];
    req.on('data', chunk => { // tacitly assuming that res.method==='POST'
      console.log(`Received ${chunk.length} bytes of data`);
      chunks.push(chunk);
    })
    req.on('end', () => {
      console.log('No more data');
      try {
        let data = JSON.parse(chunks.join(''));
        res.writeHead(200, headers);
        res.end(`${JSON.stringify(data)}\n`);
      }
      catch (err) {
        res.writeHead(400, headers);
        res.end(`${err.message}\n`);
      }
    });
    console.log(`Request: ${req.method} URL: ${req.url}`);
  });
s.listen(8080); // use curl -iX POST localhost:8080 -d <data> or -d @<file-name>
```

Request and response objects on client side

res

It is an instance of the following types (subtypes come first)

- http.IncomingMessage
- stream.Readable
- stream.Stream
- EventEmitter (it has method on)

req

It is an instance of the following types (subtypes come first)

- http.ClientRequest
- http.OutgoingMessage
- stream.Stream
- EventEmitter (it has method on)

Request and response objects on server side

req

It is an instance of the following types (subtypes come first)

- `http.IncomingMessage`
- `stream.Readable`
- `stream.Stream`
- `EventEmitter` (**it has method on**)

res

It is an instance of the following types (subtypes come first)

- `http.ServerResponse`
- `http.OutgoingMessage`
- `stream.Stream`
- `EventEmitter` (**it has method on**)

Event emitters

Introduction

- constructor `EventEmitter` exported by module `events`
- event emitters allow management of events and callbacks at a more higher level
- advantages: simpler code, expressive pattern for event driven programming
- more details later on ...

Object-oriented programming in JS

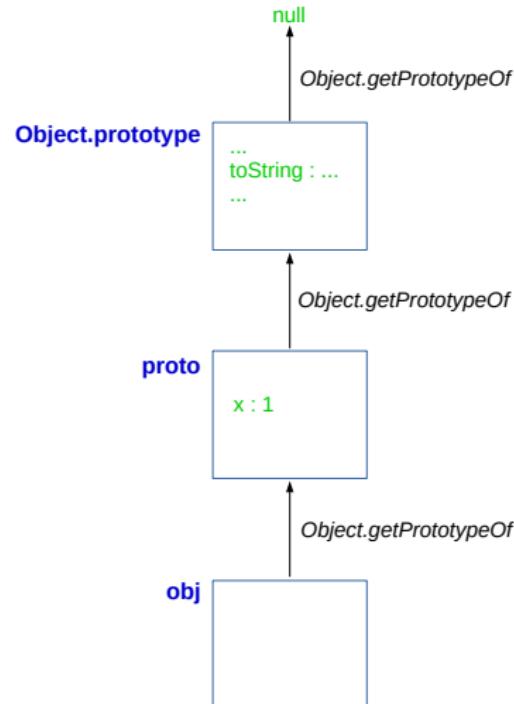
Prototype-based inheritance: the prototype chain

- an object has usually a **prototype object**
- properties are **inherited** through the prototype chain

Example with `Object.create()`

```
> let proto = {x:1}; // default prototype is 'Object.prototype'  
{ x: 1 }  
> let obj = Object.create(proto); // new object with prototype 'proto'  
{ }  
> obj.__proto__ === proto; // __proto__ property is not standard!  
true  
> Object.getPrototypeOf(obj) === proto; // Object.getPrototypeOf is standard  
true  
> obj.x;  
1  
> proto.x++;  
1  
> obj.x;  
2  
> obj.toString; // inherited from Object.prototype  
[Function: toString]
```

A pictorial view of a prototype chain



Inheritance: setting and deleting properties

Inheritance is not considered when properties are set/deleted!

```
> let proto = {x:1}; // default prototype is 'Object.prototype'  
{ x: 1 }  
> let obj = Object.create(proto); // new object with prototype 'proto'  
{ }  
> obj.x=2;  
2  
> obj;  
{ x: 2 }  
> proto;  
{ x: 1 }  
> delete obj.x;  
true  
> obj.x;  
1  
> delete obj.x;  
true  
> proto;  
{ x: 1 }
```

Property key iteration in presence of inheritance

Example

```
> let proto = {x:1}; // default prototype is 'Object.prototype'  
{ x: 1 }  
> let obj = Object.create(proto); // new object with prototype 'proto'  
{}  
> obj.y=2; // 'y' enumerable in 'obj'  
2  
> Object.defineProperty(proto,'z',{value:3}); // 'z' non-enumerable in 'proto'  
{ x: 1 }  
> Object.defineProperty(obj,'w',{value:4}); // 'w' non-enumerable in obj'  
{ y: 2 }  
> Object.getOwnPropertyNames(obj); // non-inherited properties of 'obj'  
[ 'y', 'w' ]  
> Object.keys(obj); // non-inherited enumerable properties of 'obj'  
[ 'y' ]  
> for(let p in obj) // 'in' considers also inherited enumerable properties  
    console.log(p);  
y  
x  
undefined  
> JSON.stringify(obj); // inherited and non-enumerable properties not stringified  
'{"y":2}'
```

More details on prototype-based inheritance

Constructors and prototypes

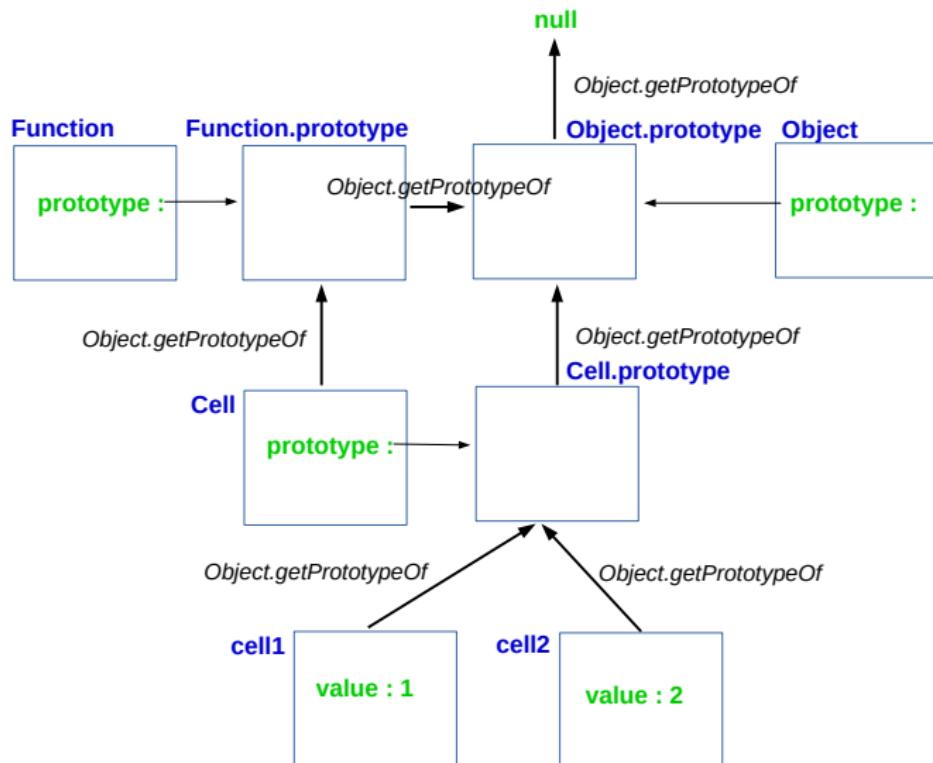
Every non-arrow function has the predefined `prototype` property

```
> function Cell(v){this.value=v;}  
undefined  
> Object.getOwnPropertyNames(Cell);  
[ 'length', 'name', 'prototype' ]
```

`prototype` is used when constructors are invoked

```
> let cell1 = new Cell(1);  
undefined  
> let cell2 = new Cell(2);  
undefined  
> Cell.prototype; // the standard prototype of all 'Cell' objects  
Cell {}  
> Object.getPrototypeOf(cell1)===Cell.prototype;  
true  
> Object.getPrototypeOf(cell2)===Cell.prototype;  
true
```

Constructors and prototypes: a pictorial view



instanceof operator

Constructors are object-oriented types

If F is a constructor (a non-arrow function), then

- o **instanceof** F checks if F .prototype is in the prototype chain of o

Example

```
> cell1 instanceof Cell;  
true  
> cell1 instanceof Object;  
true  
> cell1 instanceof Array;  
false
```

Remarks

- `prototype` is non-writable, non-enumerable and non-configurable
- arrow functions **cannot** be called as constructors

The constructor property

- the prototype property of a constructor points to an object with the constructor property
- the constructor property points back to the constructor

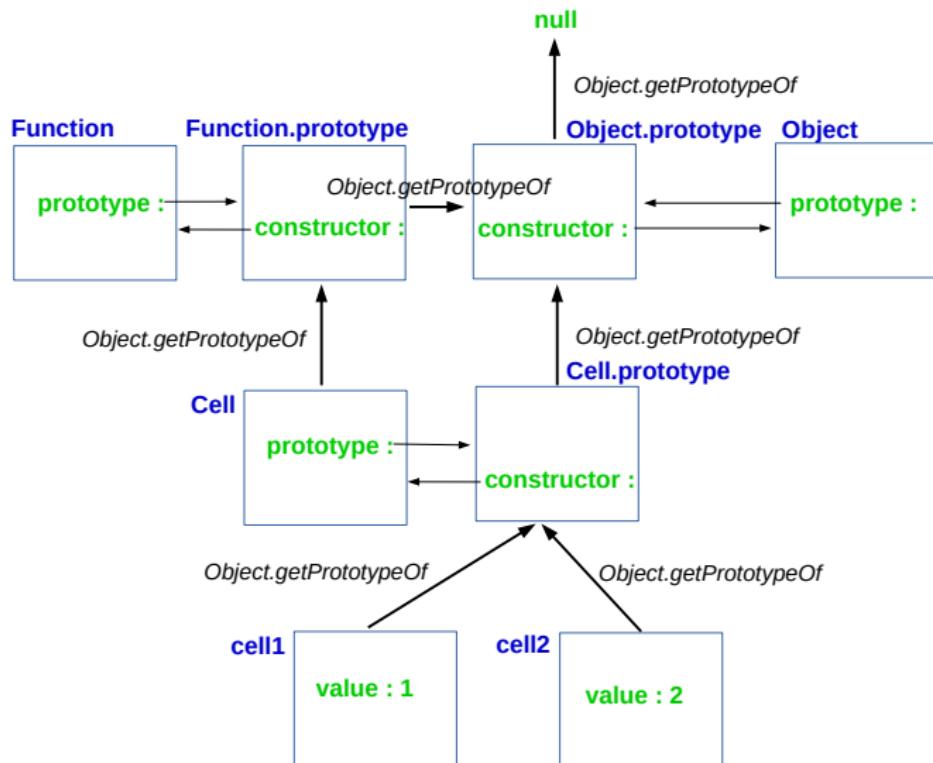
Example

```
> Cell.prototype.constructor === Cell
true
> cell1.constructor === Cell // 'constructor' inherited from 'Cell.prototype'
true
> cell2.constructor === Cell // 'constructor' inherited from 'Cell.prototype'
true
```

Remark

constructor is non-enumerable

Constructors and prototypes: a pictorial view



Object-oriented programming in JavaScript

Example 1

```
// 'Shape' works as an abstract class
function Shape() { throw new Error('Cannot create objects'); }

// method 'larger' will be inherited by 'Shape' objects
Shape.prototype.larger=
  function(other){
    return this.area()>=other.area();
}

// method 'area' expected to be defined in subclasses of 'Shape'
Shape.prototype.area=undefined;

function Square(side){this.side=side;}

// 'Square' is subclass of 'Shape'
Object.setPrototypeOf(Square.prototype,Shape.prototype)

// method 'area' will be inherited by 'Square' objects
Square.prototype.area=
  function(){
    return this.side**2;
}
```

Object-oriented programming in JavaScript

Example 2

```
function Rectangle(width,length){  
    this.width=width;  
    this.length=length;  
}  
  
// 'Rectangle' is subclass of 'Shape'  
Object.setPrototypeOf(Rectangle.prototype,Shape.prototype);  
  
// method 'area' will be inherited by 'Rectangle' objects  
Rectangle.prototype.area=  
    function(){  
        return this.width*this.length;  
    }
```

Object-oriented programming in JavaScript

Example 3

```
function MovableRectangle(width,length,lowerLeft){  
    Rectangle.call(this,width,length); // constructor called on 'this'  
    this.lowerLeft=lowerLeft;  
}  
  
// 'MovableRectangle' is subclass of 'Rectangle'  
Object.setPrototypeOf(MovableRectangle.prototype,Rectangle.prototype);  
  
// method 'move' will be inherited by 'MovableRectangle' objects  
MovableRectangle.prototype.move=  
    function(dx,dy){  
        this.lowerLeft.move(dx,dy);  
    }  
  
function Point(x,y){this.x=x; this.y=y;}  
  
// method 'move' will be inherited by 'Point' objects  
Point.prototype.move=  
    function(dx,dy){  
        this.x+=dx;  
        this.y+=dy;  
    }
```

Object-oriented programming in JavaScript

Final example

```
> let sq=new Square(4);
undefined
> let r=new Rectangle(2,3);
undefined
> let mr=new MovableRectangle(1,4,new Point(1,2));
undefined
> sq;
Square { side: 4 }
> r;
Rectangle { width: 2, length: 3 }
> mr;
MovableRectangle { width: 1, length: 4, lowerLeft: Point { x: 1, y: 2 } }
> sq.constructor==Square && r.constructor==Rectangle && mr.constructor==
    MovableRectangle;
true
> sq instanceof Square && sq instanceof Shape && sq instanceof Object;
true
> r instanceof Rectangle && r instanceof Shape && r instanceof Object;
true
> mr instanceof MovableRectangle && mr instanceof Rectangle && mr instanceof
    Shape && mr instanceof Object;
true
```

Object-oriented programming in JavaScript

Final example

```
> sq.larger(r) && r.larger(mr);
true
> sq.area();
16
> r.area();
6
> mr.area();
4
> mr.move(2,1);
undefined
> mr;
MovableRectangle { width: 1, length: 4, lowerLeft: Point { x: 3, y: 3 } }
```

More convenient syntax for OOP in ECMAScript 6

ECMAScript 6

```
class Point {  
    constructor(x, y) {this.x=x; this.y=y; }  
    move(dx, dy) {this.x+=dx; this.y+=dy; }  
}
```

ECMAScript 5

```
function Point(x, y) {this.x=x; this.y=y; }  
  
Point.prototype.move=function(dx, dy) {this.x+=dx; this.y+=dy; }
```

More convenient syntax for OOP in ECMAScript 6

ECMAScript 6

```
class MovableRectangle extends Rectangle {
    constructor(length, width, lowerLeft) {
        super(length, width);
        this.lowerLeft = lowerLeft;
    }
    move(dx, dy) {
        this.lowerLeft.move(dx, dy);
    }
}
```

ECMAScript 5

```
function MovableRectangle(width, length, lowerLeft) {
    Rectangle.call(this, width, length);
    this.lowerLeft = lowerLeft;
}
Object.setPrototypeOf(MovableRectangle.prototype, Rectangle.prototype)
MovableRectangle.prototype.move =
    function(dx, dy) {
        this.lowerLeft.move(dx, dy);
    }
```

Emitters

Details on EventEmitter

- implementation of the **observer pattern**:
 - an **emitter**, also known as **subject**, has a list of **listeners** (callbacks), also known as **observers**, for each type of event
 - when an emitter **emits an event of type *t***, then it notifies automatically its listeners (callbacks) associated with type *t* by calling them
- example: an `http.IncomingMessage` object emits events of type '**'data'**', whenever a chunk of data is available
- functions `on` and `once` allow **listeners** (callbacks) to be associated with events of a specific type
- when an event is emitted, all associated listeners are executed **synchronously** according to their **insertion order**
- listeners can be **explicitly removed**

Emitters

Difference between `on` and `once`

- `on`: the listener is always called, unless is explicitly removed
- `once`: the listener is called once, and then automatically removed

Emitters

Example 1

```
const EventEmitter = require('events');

class EmitterTest extends EventEmitter {} // ECMA6 more compact syntax

const emitter = new EmitterTest();

// registers listeners
emitter.on('a', arg=>console.log('first event listener, arg:${arg}'));
emitter.on('a', arg=>console.log('second event listener, arg:${arg}'));
emitter.once('a', arg=>console.log('third event listener, arg:${arg}'));
// emits events
emitter.emit('a', 42); // emits an event of type 'a' associated with value 42
emitter.emit('a', 0); // emits an event of type 'a' associated with value 0
```

Output

```
first event listener, arg:42
second event listener, arg:42
third event listener, arg:42
first event listener, arg:0
second event listener, arg:0
```

Emitters

Example 2

- more types of events can be emitted
- an event can be associated with an arbitrary number of arguments

```
const EventEmitter = require('events');

class EmitterTest extends EventEmitter {} // ECMA6 more compact syntax

const emitter = new EmitterTest();

// registers listeners
emitter.on('a', arg=>console.log('type a, arg:${arg}'));
emitter.on('b', ()=>console.log('type b, no arg'));
emitter.on('c', (...arg)=>console.log('type c, ${arg}'));
// emits events
emitter.emit('a', 42); // prints type a, arg:42
emitter.emit('b'); // prints type b, no arg
emitter.emit('c', 23, 42, 34.5); // prints type c, 23, 42, 34.5
```

Emitters

Selected methods

- `emitter.emit(eventType[, ...args]):` synchronously calls each of the listeners registered for ‘eventType’, in the order they were registered, passing the supplied arguments to each
- `emitter.on(eventType, listener):` adds ‘listener’ to the end of the listeners array for the events of type ‘eventType’
- `emitter.once(eventType, listener):` like method `on`, but the ‘listener’ is invoked only once and then removed from the list of listeners
- `emitter.prependListener(eventType, listener):` like method `on`, but ‘listener’ is added to the beginning of the listeners array
- `emitter.listeners(eventType):` returns a copy of the array of listeners for ‘eventType’
- `emitter.removeListener(eventType, listener):` removes ‘listener’ from the listener array for events of type ‘eventType’

Module `async`

A very useful non built-in module

- utility module for preventing multiply nested callbacks in asynchronous calls
- the code is more shallow, readable, and modularized

In a nutshell

many useful exported functions, split into two categories

- conventional functional patterns for collections
- common patterns for asynchronous control flow

Assumptions

- asynchronous functions have a single callback as the last argument
- callbacks expect an Error as their first argument
- callbacks are called just once

Module `async`

Example without `async`: writes arguments on a file

```
const { open, write, close } = require('fs'); // script fsWrite.js

function handleError(err, fd) {
    console.error(`ERROR: ${err.code}(${err.message})`);
    if (fd !== undefined) close(fd, checkError);
}

function checkError(err) {
    if (err)
        handleError(err);
}

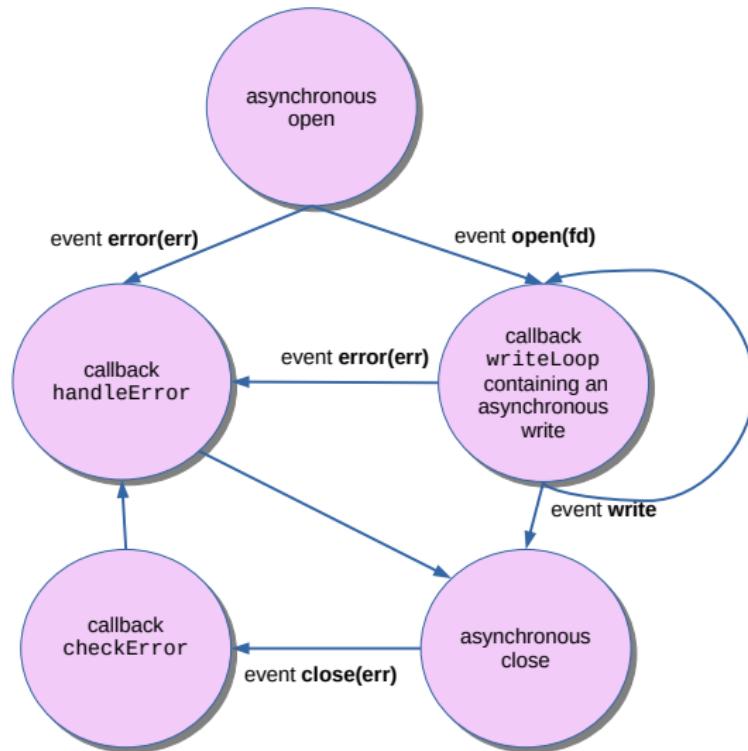
open(process.argv[2], 'w', (err, fd) => {
    let index = 3; // index of the first arg to write
    (function writeLoop(err) { // uses itself as callback
        if (err) handleError(err, fd);
        else if (index < process.argv.length)
            write(fd, process.argv[index++] + '\n', writeLoop);
        else
            close(fd, checkError);
    })(err);
});
```

Execution example:

```
$ node fsWrite.js out.txt a b c ### writes 'a' 'b' 'c' on 'out.txt'
```

Module `async`

Example without `async`



Module `async`

Solution with `async.eachSeries`

```
eachSeries(collection, iteratee, callback)
```

- collection: array or iterable
- iteratee=(item, cb) =>{ ... }: the callback to be iterated
- callback=err=>{ ... }: optional, called at the end or if an error occurs

Remarks

- items are iterated in order
- cb: hook to the next function to be called

Module `async`

Solution with `async.eachSeries`

```
const { open, write, close } = require('fs');
const { eachSeries } = require('async');
function handleError(err, fd) {
    console.error(`ERROR: ${err.code}(${err.message})`);
    if (fd !== undefined) close(fd, checkError);
}
function checkError(err) {
    if (err)
        handleError(err);
}
open(process.argv[2], 'w', (err, fd) => {
    if (err) return handleError(err);
    eachSeries(
        process.argv.slice(3), // collection, skips the first 3 elements
        (item, cb) => write(fd, item + '\n', cb), // iteratee
        err => { // final callback
            if (err) handleError(err, fd);
            else close(fd, checkError);
        });
});
```

Module `async`

Some of the most useful control flow functions in `async`

- sequential execution of asynchronous tasks

`waterfall(tasks, callbackopt)`

- parallel execution of asynchronous tasks

`parallel(tasks, callbackopt)`

- automatic management of the executions of dependent tasks

`auto(tasks, concurrencyopt, callbackopt)`

- execution of an asynchronous task within a while loop

`auto(test, iteratee, callbackopt)`

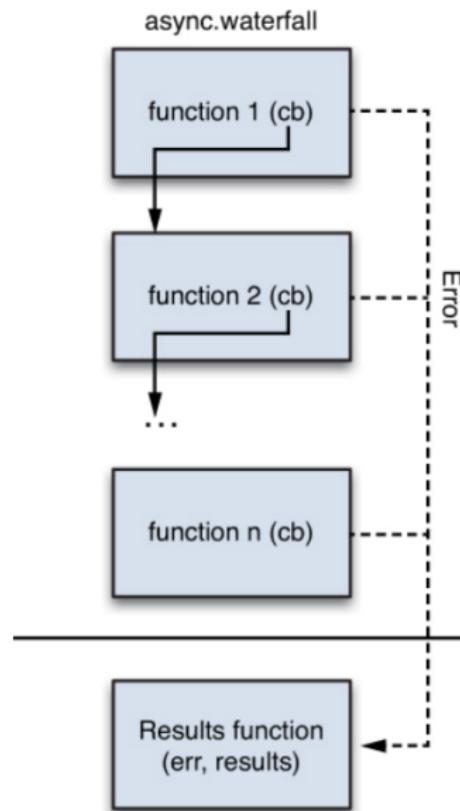
Waterfall

Use

```
waterfall([f1, ..., fn], (err, result) => {...});
```

- the tasks f_1, \dots, f_n are asynchronous functions to be run sequentially
- each function can pass to the next function an arbitrary number of results
- if f_i passes n results, then f_{i+1} has n parameters + the final callback
- the first function by the definition has only the callback parameter
- the last function f_n passes its results to the optional function
 $(err, result) => \dots$
- if f_i passes an error to their own callback, the next function is not executed, and the optional function $(err, result) => \dots$ is immediately called with the error

Waterfall



Waterfall

Example 1, with only synchronous functions

```
const { waterfall } = require('async'); // waterfall1.js
waterfall(
  [
    cb => cb(null, 'one', 'two'),           // array of functions
    (arg1, arg2, cb) => {                  // function 1
      console.log(arg1, arg2);
      cb(null, 'three');
    },
    (arg, cb) => {                        // function 2
      console.log(arg);
      cb(null, 'done');
    }
  ],
  (err, result) =>                      // results function
    err ? console.error(err.message) : console.log(result)
);
```

Remark: in case of errors, the last function is immediately called and the waterfall is interrupted

otherwise, the last function in the array passes the arguments to the last function

Waterfall

Example 2, with asynchronous functions

```
const { readFile, writeFile } = require('fs'); // waterfall2.js
const { waterfall } = require('async');

waterfall(
  [
    cb => process.argv.length<4?cb(new Error('args')):cb(null,process.argv[2]),
    (file, cb) => readFile(file, cb),
    (data, cb) => writeFile(process.argv[3],data,
      err => cb(err,Buffer.byteLength(data))),
  ],
  (err, bytes) =>
    err?console.error(err.message):console.log(`Written ${bytes} bytes`)
);

```

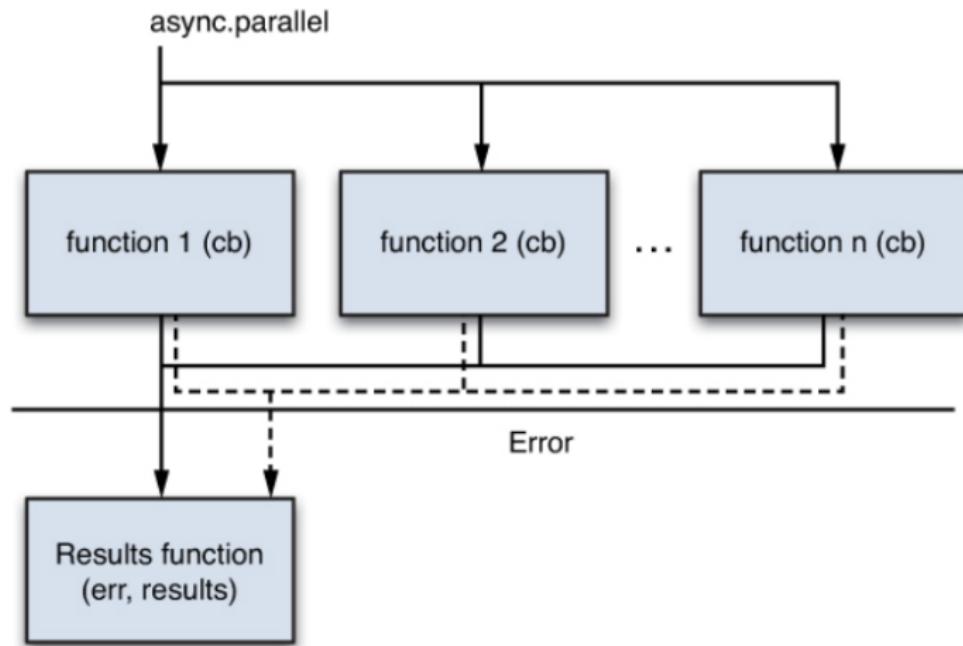
Parallel

Use

```
parallel([f1, ..., fn], (err, results) => {...});
```

- the tasks f_1, \dots, f_n can be also specified with a(possibly asynchronous) iterable or an object of functions
- the tasks are run in parallel, without waiting until the previous function has completed
- each task can complete with any number of optional result values
- if any of the functions pass an error to its callback, then
`(err, result) => {...}` is immediately called with that error
- once all the tasks have completed, the results are passed to
`(err, result) => {...}` as an array (with the corresponding order) or as an object
- in case of errors, results are collected only from the functions that completed

Parallel



Parallel

Example 1

```
const { parallel } = require('async'); // parallel.js
parallel(
  [
    cb => {
      setTimeout(() => {
        console.log('Executing one');
        cb(null, 'one', 'three');
      }, 200);
    },
    cb => {
      setTimeout(() => {
        console.log('Executing two');
        cb(null, 'two');
      }, 100);
    }
  ],
  (err, res) => err ? console.error(err.message, res) : console.log(res)
);
```

Remark: in case of errors, results are collected only from the functions that completed

Parallel

Example 2

```
const { parallel } = require('async'); // parallel2.js
parallel(
    {
        first: cb => {
            setTimeout(() => {
                console.log('Executing one');
                cb(null, 'one', 'three');
            }, 200);
        },
        second: cb => {
            setTimeout(() => {
                console.log('Executing two');
                cb(null, 'two');
            }, 100);
        }
    },
    (err, res) => err ? console.error(err.message, res) : console.log(res)
);
```

Parallel

Example 3

```
const { parallel } = require('async'); // parallel3.js
const { request } = require('http');
const url = new URL('http://212.78.1.205/IOT24-25');
const port = 1880;
const headers = {
  'Content-Type': 'application/json'
};

parallel(
  {
    one: cb => sendReq(0, 0, cb),
    two: cb => sendReq(0, 4, cb),
    three: cb => sendReq(1, 4, cb),
  },
  (err, res) => err ? console.error(err.message, res) : console.log(res)
);
```

Parallel

Example 3 (continued)

```
function sendReq(groupId, deviceId, cb) {  
    url.search = new URLSearchParams({ groupId, deviceId });  
    const req = request(url, { port, headers }, res => {  
        const chunks = [];  
        res.setEncoding('utf8');  
        res.on('data', chunk => chunks.push(chunk));  
        res.on('end', () => cb(null, chunks.join('')));  
    });  
    req.on('error', cb);  
    req.end();  
}
```

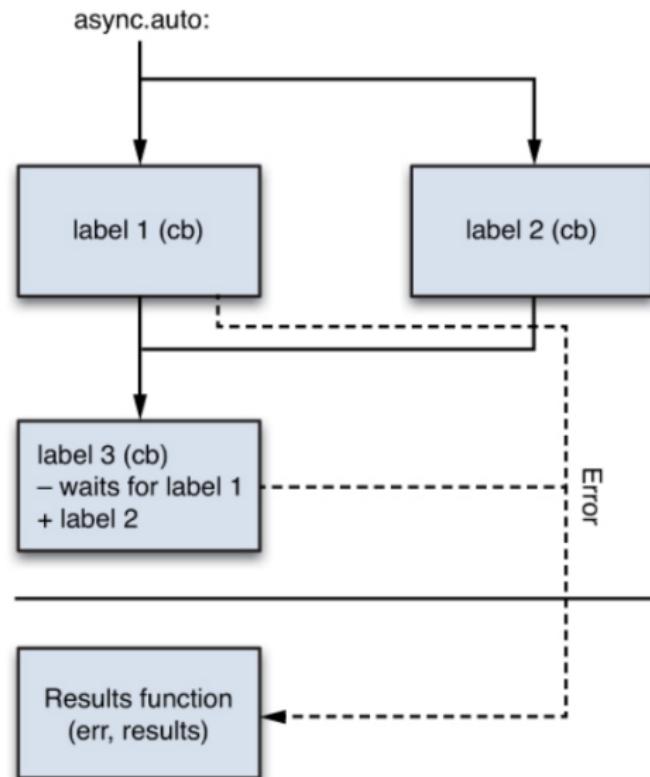
Auto

Use

```
auto({l1:f1, ... ln:[d1,...,dk,fn]}, (err, results) => {...});
```

- the tasks f_1, \dots, f_n are defined within an object
- each of its properties is either a function or an array of dependencies, with the task itself the last item in the array
- the object's key of a property serves as the name of the task defined by that property and is used when specifying dependencies for other tasks
- dependencies cannot be mutually recursive
- each task receives one or two arguments: (1) a results object, containing the results of the previously executed functions, only passed if the task has any dependencies (2) a callback to be called when finished, with the error and the result of the function's execution
- an optional final callback is called when all the tasks have been completed. Results are always returned; however, if an error is received, no further tasks will be performed, and the results object will only contain partial results

Auto



Auto

Example 1

```
const { readFile, writeFile } = require('fs'); // autol.js
const { auto } = require('async');
auto(
  {
    check: cb =>
      process.argv.length < 4 ? cb(new Error('args')) : cb(null, process.argv[2]),
    read: ['check', (results, cb) => readFile(results.check, 'utf8', cb)],
    write: ['read', (results, cb) => writeFile(process.argv[3], results.read, cb)],
  },
  (err, res) => err ? console.error(err.message, res) : console.log(res)
);
```

Auto

Example 2

```
const { auto } = require('async'); // auto2.js
auto({
  getData: cb => {
    console.log('in getData');
    cb(null, 'data', 'converted to array');
  },
  makeFolder: cb => {
    console.log('in makeFolder');
    cb(null, 'folder');
  },
  writeFile: ['getData', 'makeFolder', (results, cb) => {
    console.log('in writeFile', results);
    cb(null, 'filename');
  }],
  emailLink: ['writeFile', (results, cb) => {
    console.log('in emailLink', results);
    cb(null, { file: results.writeFile, email: 'user@example.com' });
  }]
},
  (err, res) => err ? console.error(err.message, res) : console.log(res)
);
```

Whilst

Use

```
whilst(cb=>{ ...cb(err,test)...}, cb => { ...cb(err,result)...}, (err,  
result)=>{ ...});
```

- it executes asynchronous tasks sequentially as a while statement
- `cb=>{ ...cb(err,test)...}` is the asynchronous truth test to perform before each execution of iteratee
- the iteratee `cb => { ...cb(err,result)...}` is the asynchronous function which is called each time test passes
- `(err,result)=>{ ...}` is called after the test function has failed and repeated execution of iteratee has stopped. The callback will be passed an error and any arguments passed to the final iteratee's callback

Whilst

Example 1

```
const { whilst } = require('async');
let count = 0;
whilst(
  cb => cb(null, count < 5),
  cb => {
    console.log(++count);
    setTimeout(() => cb(null, count), 1000);
  },
  (err, n) => err ? console.error(err) : console.log(`seconds elapsed: ${n}`)
);
```

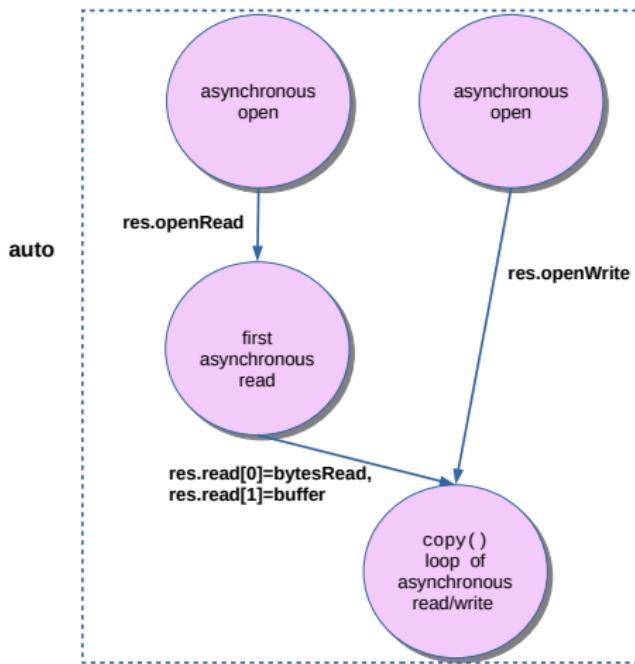
Whilst

Example 2

```
const { whilst } = require('async');
const { request } = require('http');
const url = new URL('http://212.78.1.205/IOT24-25');
const port = 1880;
const headers = {
  'Content-Type': 'application/json'
};
const maxDevices = 10;
let deviceId = 0;
whilst(
  cb => cb(null, deviceId < maxDevices),
  cb => sendReq(0, deviceId, (err, res) => {
    if (!err) console.log(res); cb(err, ++deviceId); }),
  (err,n) => err?console.error(err):console.log('read ${n} devices')
);
```

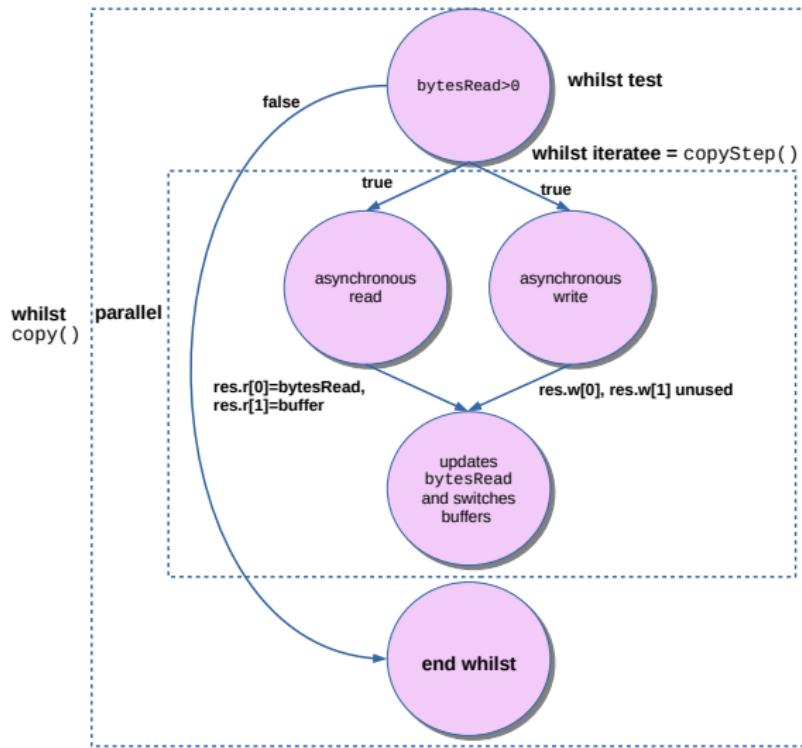
Advanced example with auto, whilst and parallel

Copy files of arbitrary size with two buffers



Advanced example with auto, whilst and parallel

Copy files of arbitrary size with two buffers



Advanced example with auto, whilst and parallel

Copy files of arbitrary size with two buffers

```
const { open, read, write, close } = require('fs'); // auto3.js
const { auto, parallel, whilst } = require('async');
const BUFF_SIZE = 2 ** 20;
let rbuf = Buffer.allocUnsafe(BUFF_SIZE); // buffers for reading and writing
let wbuf = Buffer.allocUnsafe(BUFF_SIZE);
auto(
{
    openRead: cb => open(process.argv[2], 'r', cb),
    openWrite: cb => open(process.argv[3], 'w', cb),
    read: ['openRead', (res, cb) => read(res.openRead, wbuf, cb)],
    copy: ['read', 'openWrite', copy],
},
checkErrorAndClose
);
function checkErrorAndClose(err, res) {
    if (err) console.error(`ERROR: ${err.code} (${err.message})`);
    if (res !== undefined) {
        tryClose(res.openRead);
        tryClose(res.openWrite);
    }
}
function tryClose(fd) { if (fd !== undefined) close(fd, checkErrorAndClose); }
```

Advanced example with auto, whilst and parallel

Copy files of arbitrary size with two buffers

```
function copy(res, cb) {
  let bytesRead = res.read[0]; // res.read[0]=bytesRead, res.read[1]=buffer
  whilst(
    cb => cb(null, bytesRead > 0), // test, no error occurs
    copyStep,                      // iteratee
    cb                            // final callback
  ); // end whilst
  function copyStep(cb) { // nested in function copy
    ...
  }
}
```

Advanced example with auto, whilst and parallel

Copy files of arbitrary size with two buffers

```
function copyStep(cb) { // nested in function copy
    parallel(
        {
            r: cb => read(res.openRead, rbuf, cb),
            w: cb => write(res.openWrite, wbuf, 0, bytesRead, cb)
        },
        (err, res) => {
            if (err)
                cb(err);
            else {
                bytesRead = res.r[0]; // res.r[0]=bytesRead, res.r[1]=buffer
                const tmp = rbuf; // switches buffers
                rbuf = wbuf, wbuf = tmp;
                cb();
            }
        }
    ); // end parallel
}
```

Promises

History and motivations

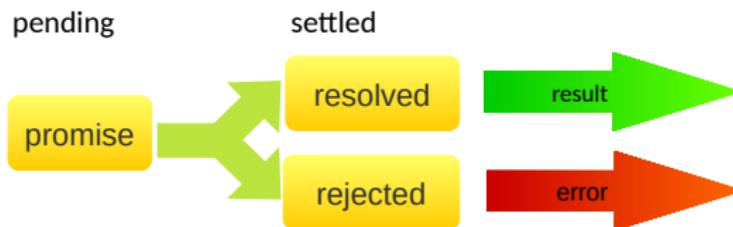
- based on a programming model for asynchronous computing originally proposed in the 70s
- standardized in ECMAScript 6
- better control flow for asynchronous computing
 - avoid the “callback hell”
 - better support for error handling
 - no lost events

Promises

Overview

Aim: a promise models the result associated with an asynchronous operation
three possible states of the asynchronous operation:

- pending (=not settled) or settled: not completed yet or completed
- resolved (=fulfilled): settled, completed successfully with a value
- rejected: settled, failed with an error



A simple example

fs.open and promises

```
const { open } = require('fs');
function openFile(fname) {
    return new Promise(
        (res, rej) => open(fname, 'w', (err, fd) => err ? rej(err) : res(fd))
    );
}
const fd = openFile(process.argv[2] || 'out.txt');
console.log(fd); // prints Promise { <pending> }
```

Remark

- `res` is called with `v` if the promise is resolved with `v`
- `rej` is called with `err` if the promise is rejected with `err`
- the `Promise` constructor can create already resolved promises

```
> Promise.resolve(42);
Promise { 42, ... }          // promise already resolved
> Promise.reject(new Error('error'));
Promise { <rejected> 42, ... } // promise already rejected
> Uncaught Error: error
```

Methods then/catch/finally

Motivation and use

specification of **resolve/reject reactions** (or **handlers**) for

- **asynchronously** accessing the value of a **resolved** promise
- **asynchronously** managing the error of a **rejected** promise

Use:

`p.then(resolve-reaction, reject-reaction)`

`p.catch(reject-reaction)` equivalent to `p.then(undefined, reject-reaction)`

`p.finally(finally-reaction)` equivalent to

`p.then(finally-reaction, finally-reaction)`

- `resolve-reaction` called with argument *v* when *p* fulfilled with value *v*
- `reject-reaction` called with argument *err* when *p* rejected with error *err*
- `finally-reaction` called anyway with no argument

Methods then/catch/finally

Example

```
function printFd(fd) { console.log('Descriptor: ${fd}'); }

function printErr(err) { console.error(err.message); }

// more idiomatic usage
openFile(process.argv[2] || 'out.txt').then(printFd).catch(printErr);

// equivalent code
openFile(process.argv[2] || 'out.txt').then(printFd, printErr);
```

More on reactions

A reaction is always scheduled, but asynchronously
even when registered **after** its promise has been already settled

```
const pr = Promise.resolve(42);                      // a resolved promise

// pr already resolved with 42
pr.then(val => console.log('Value: ${val}'));      // no reject reaction needed

pr.then(val => console.log('Same value: ${val}')); // no reject reaction needed

console.log('Still need to call the reaction...');
```

Result

Still need to call the reaction...

Value: 42

Same value: 42

Promise chaining

Methods `then/catch/finally` return new promises

Example:

```
const pr=Promise.resolve(42);

pr.then(val=>{console.log(val); return val;}).then(console.log);
```

Result

```
42
42
```

Typical example of use

```
openFile(process.argv[2] || 'out.txt').
  then(printFd).
  catch(printErr).
  finally(() => console.log('done'));
```

More details on the behavior of then/catch/finally

Rules

- reactions are called starting from the next event loop, even when the promise is already settled
- the returned promise is always initially pending, it will be eventually settled starting from the next event loop

```
const pr=Promise.resolve(42);
console.log(pr);
const next=pr.then(console.log);
console.log(next);
```

Result

```
Promise { 42 }
Promise { <pending> }
42
```

Promises and the microtask queue

Rule

- promise reactions are handled with a **promise microtask queue**
- the promise microtask queue has higher priority than the timer queue
- the promise microtask queue has lower priority than the `nextTick` microtask queue

Promises and the microtask queue

Example

```
setImmediate(() => console.log('setImmediate'));
setTimeout(() => console.log('timeout'));
const pr = Promise.resolve(42);
queueMicrotask(() => console.log('microtask 1'));
const next1 = pr.then(() => console.log('then 1'));
const next2 = next1.then(() => console.log('then 2'));
queueMicrotask(() => console.log('microtask 2'));
process.nextTick(() => console.log('nextTick'));
console.log(pr, next1, next2);
```

Results:

```
Promise { 42 } Promise { <pending> } Promise { <pending> }
nextTick
microtask 1
then 1
microtask 2
then 2
timeout
setImmediate
```

A more complex example of chaining

Simplified version, not always the file is properly closed

```
const { open, write, close } = require('fs');
function openFile(fname) {
    return new Promise((res, rej) =>
        open(fname, 'w', (err, fd) => err ? rej(err) : res(fd)));
}
function writeFile(fd) {
    return new Promise((res, rej) =>
        write(fd, 'test\n', err => err ? rej(err) : res(fd)));
}
function closeFile(fd) {
    return new Promise((res, rej) =>
        close(fd, err => err ? rej(err) : res()));
}
function handleError(err) {
    console.error(err.message);
}

openFile(process.argv[2] || 'out.txt').
    then(writeFile).
    then(closeFile).
    catch(handleError).
    finally(() => console.log('done'));
```

A more complex example of chaining

Full version, the file is always properly closed

```
const { open, write, close } = require('fs');
function openFile(fname) {
    return new Promise((res, rej) =>
        open(fname, 'w', (err, fd) => err ? rej({ err }) : res(fd)));
}
function writeFile(fd) {
    return new Promise((res, rej) =>
        write(fd, 'test\n', err => err ? rej({ err, fd }) : res(fd)));
}
function closeFile(fd) {
    return new Promise((res, rej) =>
        close(fd, err => err ? rej({ err }) : res()));
}
function handleError(rej) {
    console.error(rej.err.message);
    if (rej.fd != undefined) return closeFile(rej.fd).catch(handleError);
}

openFile(process.argv[2] || 'out.txt').
    then(writeFile).
    then(closeFile).
    catch(handleError).
    finally(() => console.log('done'));
```

Promisified functions

Example

```
const fs = require('fs');
const { promisify } = require('util');
const [open, write, close] = [promisify(fs.open), promisify(fs.write), promisify(
  fs.close)];
```

- open, write, close return a promise
- if the callback of the asynchronous function has more non-error parameters, then the promise is resolved with an object where the property names are those of the parameters

Example: promises of write are resolved with objects with property names bytesWritten and buffer

Remark: the promisified version of some functions in fs are already defined in require('fs').promises (or require('fs/promises')), but not write(), read() and close()

Reaction types

Rules on reaction types

`p.then(resolve-reaction, reject-reaction)`

- if a called reaction has not type '`function`', then the returned promise will be resolved/rejected as `p`

For this reason `p.catch(reject-reaction)` is used as shorthand for
`p.then(undefined, reject-reaction)`

- if the return type of a called reaction is a non-promise value `v`, then the returned promise will be resolved with `v`
- if a called reaction throws `err`, then the returned promise will be rejected with `err`
- if a reaction returns a promise `p'`, then the returned promise will resolve as `p'`

Promise.all (iterable)

Useful for synchronization of more promises

- iterable: a sequence of promises
- if all promises are resolved, then `Promise.all (iterable)` is resolved with an array of their values
- if one promise is rejected, then `Promise.all (iterable)` is rejected with the reason of the promise rejected first

Example:

```
const { open } = require('fs').promises;

const fname1 = process.argv[2] || 'in1.txt';
const fname2 = process.argv[3] || 'in2.txt';

Promise.all([open(fname1, 'r'), open(fname2, 'r')]).
  then(console.log).
  catch(console.error);
```

See also `Promise.allSettled()` (ES2020)

Promise.race(iterable)

Useful to race promises

- iterable: a sequence of promises
- Promise.race(iterable) is settled in the same way as the first promise settled in iterable

Example:

```
const {open}=require('fs').promises;

const fname1=process.argv[2]||'in1.txt';
const fname2=process.argv[3]||'in2.txt';

Promise.race([open(fname1, 'r'), open(fname2, 'r')]).then(console.log).catch(console.error);
```

async/await (ECMAScript 2017)

Motivations

- transparent use of promises
- synchronous-style programming with asynchronous functions

async/await (ECMAScript 2017)

Rules

- the use of `await` allows automatic handling of promises by pausing the execution of the enclosing `async` function
 - if the promise is resolved, then the execution of the `async` function is resumed and the value of the `await` expression is that of the fulfilled promise
 - if the promise is rejected, then the execution of the `async` function is resumed and the `await` expression throws the rejected value
- `await` can only be used inside an `async` function within regular JavaScript code
- `await` can be used on its own with ES modules
- if a function is `async`, then its return value will be a promise even if no promise-related code appears in the body of the function; if it throws an exception, then it will return a promise rejected with that exception

async/await (ECMAScript 2017)

Example 1 with a CommonJS module

```
'use strict';

const { readFile } = require('fs/promises'); // CommonJS module

async function read(fname) {
  const data = await readFile(fname, { encoding: 'utf-8' });
  console.log(data);
}

read(process.argv[2] || 'test.txt');
console.log('reading');
```

Result

```
reading
this
is
a
test
```

async/await (ECMAScript 2017)

Example 2 with a CommonJS module

```
const fs = require('fs'); // CommonJS module
const { promisify } = require('util');
const SIZE = 2 ** 10;
const [open, read] = [promisify(fs.open), promisify(fs.read)];

async function openThenRead() {
  const fd = await open(process.argv[2] || 'test.txt', 'r');
  const buf = Buffer.allocUnsafe(SIZE);
  const res = await read(fd, buf);
  console.log(buf.toString('utf8', 0, res.bytesRead));
}

openThenRead();
```

async/await (ECMAScript 2017)

Same example with an ES6 module

```
import fs from 'fs'; // ES6 module
import { promisify } from 'util';
const SIZE = 2 ** 10;
const open = promisify(fs.open);
const read = promisify(fs.read);
const fd = await open(process.argv[2] || 'test.txt', 'r');
const buf = Buffer.allocUnsafe(SIZE);
const res = await read(fd, buf);
console.log(buf.toString('utf8', 0, res.bytesRead));
```

Remark: `await` is only valid in

- async functions
- the top level bodies of ES6 modules

async/await (ECMAScript 2017)

Example with try-catch

```
const fs = require('fs'); // CommonJS module
const { promisify } = require('util');
const SIZE = 2 ** 10;
const [open, read] = [promisify(fs.open), promisify(fs.read)];

async function openThenRead() {
  try {
    const fd = await open(process.argv[2] || 'test.txt', 'r');
    const buf = Buffer.allocUnsafe(SIZE);
    const res = await read(fd, buf);
    console.log(buf.toString('utf8', 0, res.bytesRead));
  }
  catch (err) { console.error(err.message); }
}

openThenRead();
```

async/await (ECMAScript 2017)

Example with try-catch and loop

```
const fs = require('fs'); // CommonJS module
const { promisify } = require('util');
const SIZE = 2 ** 10;
const [open, read, write] =
  [promisify(fs.open), promisify(fs.read), promisify(fs.write)];

async function openThenRead() {
  try {
    const fd = await open(process.argv[2] || 'test.txt', 'r');
    const buf = Buffer.allocUnsafe(SIZE);
    let res;
    do {
      res = await read(fd, buf);
      await write(process.stdout.fd, buf, 0, res.bytesRead);
    } while (res.bytesRead > 0);
  }
  catch (err) { console.error(err.message); }
}

openThenRead();
```

async/await (ECMAScript 2017)

Pros and cons

- simpler and more readable code with synchronous style
- more complex semantics, subtle errors, less efficient in some cases

Example 1

```
import fs from 'fs'; // ES6 module
import { promisify } from 'util';
const SIZE = 2 ** 10;
const open = promisify(fs.open);
const read = promisify(fs.read);
const fd = await open(process.argv[2] || 'test.txt', 'r'); // code is paused
console.log(fd); // prints a number, implicit promise already resolved
const buf = Buffer.allocUnsafe(SIZE);
const res = read(fd, buf); // remark: no await is used, code is not paused
console.log(res); // prints a pending implicit promise
```

async/await (ECMAScript 2017)

Pros and cons

- simpler and more readable code with synchronous style
- more complex semantics, subtle errors, less efficient in some cases

Example 1

```
import fs from 'fs'; // ES6 module
import { promisify } from 'util';
const SIZE = 2 ** 10;
const open = promisify(fs.open);
const read = promisify(fs.read);
const fd = await open(process.argv[2] || 'test.txt', 'r'); // code is paused
console.log(fd); // prints a number, implicit promise already resolved
const buf = Buffer.allocUnsafe(SIZE);
const res = await read(fd, buf); // code is paused again
console.log(res); // prints an object, implicit promise already resolved
```

async/await (ECMAScript 2017)

Pros and cons

- simpler and more readable code with synchronous style
- more complex semantics, subtle errors, less efficient in some cases

Example 2

Less efficient and unnecessarily sequential code:

```
// ES module
const response1 = await request(server1); // code is paused
const response2 = await request(server2); // code is paused again
aggregate(response1, response2);
```

More efficient code with explicit use of Promise.all:

```
// ES module
const [response1, response2] =
  await Promise.all(request(server1), request(server2)); // parallel requests
aggregate(response1, response2);
```

Async iterators and generators

Overview

- async iterators allow iteration over promises
- async generators are functions which return async iterators
- iteration on async iterators can be performed with the statement

```
for await (const/let ... of ...) {...}
```

- the statement can only be used in ES6 modules or async functions

Async iterators and generators

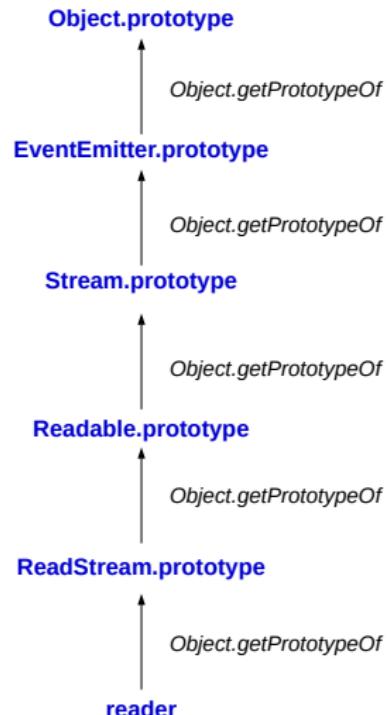
Example

```
function asyncFun(val, timeout) {
  return new Promise(res => setTimeout(res, timeout, val));
}
function* generate() { // async is needed only if await is used in the body
  yield asyncFun(1, 0);
  yield asyncFun(2, 1000);
  yield asyncFun(3, 1500);
}
(async function () {
  for await (const val of generate())
    console.log(val);
})()
```

Readable streams

Type hierarchy

```
> const { createReadStream } = require('fs');
undefined;
> const reader = createReadStream('./in.txt');
undefined;
```



Readable streams

Operation modes

two operation modes: **flowing** and **paused**

- flowing mode: data **read automatically** and provided as quickly as possible with `EventEmitter` events
- paused mode: the `read()` method **must be called explicitly** to read chunks of data
- all readable streams **begin in paused mode**
- they can be **switched to flowing mode** by (1) adding a '`data`' event handler, (2) calling the `resume()` or `pipe()` methods
- they can **switch back to paused mode** by (1) calling the `pause()` method if there are no pipe destinations, (2) calling the `unpipe()` method if there are pipe destinations
- the attribute `highWaterMark` determines the size of the buffer used by the stream, and, hence of the read chunks

Readable streams

Example

```
const { createReadStream } = require('fs');

const reader = createReadStream(process.argv[2], { highWaterMark: 2 ** 4 });

reader.once('open', () => console.log('open'));

reader.on('data', chunk => { // switches from paused to flowing mode
    console.log('data: ', chunk.toString());
    reader.pause(); // switches back to paused mode
    setTimeout(() => reader.resume(), 1000); // switches to flowing mode after 1 second
});

reader.on('pause', () => console.log('pause'));

reader.on('resume', () => console.log('flowing mode'));

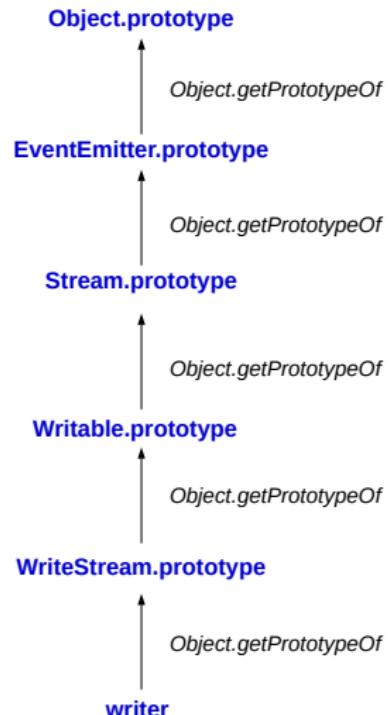
reader.once('end', () => console.log('end'));

reader.once('close', () => console.log('close'));
```

Writable streams

Type hierarchy

```
> const { createWriteStream } = require('fs');
undefined;
> const writer = createWriteStream('./out.txt');
undefined;
```



Writable streams

- the basic operations on writable streams are `write()` and `end()`
- the attribute `highWaterMark` determines the size of the buffer of the stream

Simple example

```
const { createWriteStream } = require('fs');
const writer = createWriteStream('./out.txt');
writer.write('some data\n');
writer.write('some more data\n');
writer.end('done\n'); // optionally writes and signals no other data is written
```

Selection of events

- '`finish`': emitted after `end()` has been called, and all data has been flushed
- '`close`': emitted after '`finish`', when the stream has been closed
- '`drain`': emitted when it is appropriate to write data, if a call to `write()` has returned false because the buffer is full

Writable streams

More on draining

- if the stream is not draining, then calls to `write(chunk)` return false and chunk is buffered
- once all currently buffered chunks are drained, the '`'drain'`' event will be emitted
- if `write()` returned false, no more chunks should be written until the '`'drain'`' event is emitted, to avoid memory/performance/security problems

A safe use of a writable stream

```
function write(data, cb) {  
  if (!stream.write(data)) {  
    stream.once('drain', cb);  
  } else {  
    process.nextTick(cb);  
  }  
}  
write('hello', () => {console.log('write completed, do more writes now');});
```

Write streams

Another example with the 'drain' event

```
const N = 100;
const { createWriteStream } = require('fs');
const writer = createWriteStream(process.argv[2], { highWaterMark: 2 ** 4 });

writer.once('finish', () => console.log('finish'));
writer.once('close', () => console.log('close'));

function write(i) {
    if (i >= N)
        writer.end(); // ends writing
    else {
        if (!writer.write(i + '\n')) {
            console.log('waiting to drain');
            writer.once('drain', () => write(i + 1)); // buffer limit exceeded
        }
        else
            write(i + 1);
    }
}

write(0); // prints 0,...,N-1 on the write stream
```

Pipes

Pipes are used to connect readable to writable streams

Example:

```
'use strict';

const { createReadStream, createWriteStream } = require('fs');

const reader = createReadStream(process.argv[2]);

const writer = process.argv.length > 3 ?
  createWriteStream(process.argv[3]) : process.stdout;

reader.pipe(writer); // copies reader to writer
```

Pipes

Pipeline

pipeline is recommended for gracefully handling errors and avoiding memory leaks

```
const { createReadStream, createWriteStream } = require('fs');

const { pipeline } = require('stream');

const reader = createReadStream(process.argv[2]);

const writer = process.argv.length > 3 ?
  createWriteStream(process.argv[3]) : process.stdout;

pipeline(reader, writer, err => { if (err) console.error('Copy failed', err); });
```

Pipes

Pipeline

- the `pipeline()` operator accepts a variable number of arguments
- except for the last callback argument, all other arguments can be:
 - readable and writable streams
 - (async) iterables (only as sources)
 - (async) generator functions

Pipes

Pipeline

example of pipeline with an async generator and a writable stream (stdout)

```
const MAX = 10;
const { pipeline } = require('stream');
function asyncFun(val, timeout) {      // an async operation
    return new Promise(res => setTimeout(res, timeout, val));
}
function* generate({ signal }) { // generates ints from 0 to MAX-1
    for (let i = 0; i < MAX; i++)
        yield asyncFun(i + '\n', 1000);
}
pipeline(
    generate,
    process.stdout,
    err => { if (err) console.error(err); });
```

Pipes

Pipeline

example of pipeline with an iterable (an array), an async generator and a writable stream (stdout)

```
const MAX = 21;
const { pipeline } = require('stream');
const data = require('./data.json'); // a JSON array of integers
function asyncFun(val, timeout) { // an async operation
    return new Promise(res => setTimeout(res, timeout, val));
}
async function* lessThanMax(source, { signal }) { // keeps values < MAX
    for await (const chunk of source) {
        if (chunk < MAX)
            yield asyncFun(chunk + '\n', 1000);
    }
}
pipeline(
    data,
    lessThanMax,
    process.stdout,
    err => { if (err) console.error(err); });

```

Transform streams

Overview

- they are **duplex streams**
- used in pipes to relate the output to the input
- they implement both readable and writable streams
- typical examples: compression/encryption/conversion/filtering of data streams

```
pipeline(stream1, transform, stream2)
```

- `transform` is a **writable** stream for `stream1`
- `transform` is a **readable** stream for `stream2`

Transform streams

Same example as before, but with a transform instead of a generator

```
const MAX = 21;
const { pipeline, Transform } = require('stream');
const data = require('./data.json'); // a JSON array of integers
const lessThanMax = new Transform({ // keeps values < MAX
    objectMode: true, // chunks are not exclusively of type string/buffer
    transform(chunk, encoding, callback) {
        callback(null, chunk < MAX ? chunk + '\n' : undefined);
    }
});
pipeline(
    data,
    lessThanMax,
    process.stdout,
    err => { if (err) console.error(err); });
```

- `transform(chunk, encoding, callback)` should be redefined in a transform object
- the second argument of the callback is the chunk to be pushed in the buffer of the readable portion of the transform

Transform streams

Example of conversion to upper case

```
const { createReadStream, createWriteStream } = require('fs');
const { pipeline, Transform } = require('stream');
const reader = createReadStream(process.argv[2]);
const writer = process.argv.length > 3 ?
    createWriteStream(process.argv[3]) : process.stdout;
const toUpper = new Transform({
    transform(chunk, encoding, callback) {
        this.push(chunk.toString().toUpperCase());
        callback();
    },
    flush(callback) { // called when there is no more written data to be consumed
        this.push('\n'); // inserts a new line at the end
        callback();
    }
});
pipeline(reader, toUpper, writer, err => {
    if (err) console.error('Transformation failed', err);
});
```

Transform streams

Another example: extracts city and temperature from JSON and converts back to JSON

```
const { createReadStream, createWriteStream } = require('fs');
const { pipeline, Transform } = require('stream');
const reader = createReadStream(process.argv[2]);
const writer = process.argv.length > 3 ?
  createWriteStream(process.argv[3]) : process.stdout;
const valueFromJSON = new Transform({
  construct(callback) { this.data = ''; callback(); }, // constructor redefined
  transform(chunk, encoding, callback) {
    this.data += chunk; callback();
  },
  flush(callback) {
    try {
      const { city, temperature } = JSON.parse(this.data);
      this.push(JSON.stringify({ city, temperature }));
      callback(null, '\n');
    }
    catch (err) { callback(err); }
  }
});
pipeline(reader, valueFromJSON, writer, err => {
  if (err) console.error('Transformation failed', err);
});
```

Promisified pipelines

Example

```
const MAX = 21;
const { pipeline } = require('stream/promises');
const { Transform } = require('stream');
const data = require('./data.json'); // a JSON array
const lessThanMax = new Transform({ // keeps values < MAX
    objectMode: true, // chunks are not exclusively of type string/buffer
    transform(chunk, encoding, callback) {
        callback(null, chunk < MAX ? chunk + '\n' : undefined);
    }
});
(async function () {
    await pipeline(
        data,
        lessThanMax,
        process.stdout);
})().catch(console.error);
```

CommonJS Modules

CommonJS JavaScript modules

- they can be imported with the predefined function `require (id)`
id is a module name or a (absolute/relative) path (for local files)
- they can be exported by using the predefined objects `module` or `exports`
 - `module.exports` refers to an initially empty object
 - `exports==module.exports` initially holds

Two ways to export features

- (more common) `module.exports` is an object where each property is an exported feature; examples:
 - a function (e.g. `fs.open`)
 - a constant or many constants (e.g. `fs.constants`)
 - a constructor (e.g. `fs.ReadStream`)
- (less common) `module.exports` is a constructor;
example: `const EventEmitter=require('events');`

CommonJS Modules

Example: how to export definitions

```
// file points.js
class Point {
    constructor(x, y) { this.x = x; this.y = y; }
    move(dx, dy) { this.x += dx; this.y += dy; }
}
exports.Point = Point; // or 'module.exports.Point=Point'
exports.origin = new Point(0, 0); // or 'module.exports.Point=new Point(0, 0)'
```

Example: how to import definitions

```
// file use_points.js
const { Point, origin } = require('./points.js');
let p = new Point(1, 2);
p.move(1, 0);
console.log(p, origin);
```

Output:

```
Point { x: 2, y: 2 } Point { x: 0, y: 0 }
```

CommonJS Modules

Example: how to export a single constructor

```
// file points2.js
class Point {
    static origin = new Point(0, 0); // class variable
    constructor(x, y) { this.x = x; this.y = y; }
    move(dx, dy) { this.x += dx; this.y += dy; }
}
module.exports = Point; // this module only exports constructor Point
```

Example: how to import a single constructor

```
// file use_points2.js
const Point = require('./points2.js'); // import local file with relative
path
let p = new Point(1, 2);
p.move(1, 0);
console.log(p, Point.origin);
```

Output:

```
Point { x: 2, y: 2 } Point { x: 0, y: 0 }
```

npm (Node Package Manager)

Three main components

- the Web site: for npm user accounts
- the Command Line Interface (CLI): see the next slide
- the registry: a large public database of JavaScript packages at
<https://registry.npmjs.org>

Modules versus packages

- *module*: any file or directory in a `node_modules` directory that can be loaded with `require`
- *package*: a file or directory specified by a `package.json` file

Remark

`package.json` can be automatically and interactively generated with `npm init`

npm (Node Package Manager)

A selection of commands from CLI

npm init (*generates package.json*)

npm search (*searches the registry for packages*)

npm install (*installs a package, and any packages that it depends on*)

npm update (*update all the packages listed to the latest version*)

npm link (*creates a symbolic link from a globally-installed package*)

npm uninstall (*completely uninstalls a package*)

npm publish (*publishes a package to the registry so that it can be installed by name*)

npm help (*shows documentation pages*)

Folder-based modules

Rules

Folder-based modules can consist of more files

- required files are collected in a specific subfolder of `node_modules` with the same name of the module
- the subfolder usually contains the file `package.json`

```
{  
  "name": "points",  
  "version": "1.0.0",  
  "description": "A simple class for 2D points",  
  "main": "index.js",  
  "scripts": { "test" : "node test.js" },  
  "author": "Davide Ancona",  
  "private": "true"  
}
```

- if no `package.json` or `main` property in `package.json` are provided, then `index.js` or `index.node` are searched

Module/package search

Rules

Search based on the following order:

- ① built-in modules are searched (as `fs` or `http`)
- ② if a path is specified, then
 - ① if an extension is given, then the file with that path is searched
 - ② if the previous attempt fails or no extension is given, then extensions `.js`, `.json`, and `.node` are tried in order
 - ③ if the previous attempt fails, then a folder-based module is searched
 - ④ if the previous attempt fails, then an error is thrown

if no path is specified, then

- ① look up starts from the `node_modules` subfolder of the current folder up to the tree root
- ② if the previous attempt fails, then standard default locations are searched (as `/usr/lib`, `/usr/local/lib`)
- ③ if the previous attempt fails, then an error is thrown

Remark: imported modules are **private** to the modules that include them

Module caching, versions and dependency

Caching and versions

- every module is loaded and initialized just once even when it is required by more modules
- several versions of the same module can coexist in different node_modules subfolders

Cyclic dependencies are allowed!

```
// file rec-mod1.js
const m2=require('./rec-mod2');
exports.val=m2.val+1;
console.log('mod1.val=${exports.val}'');

// file rec-mod2.js
const m1=require('./rec-mod1');
exports.val=m1.val+1;
console.log('mod2.val=${exports.val}'');
```

Output:

```
node --use-strict rec-mod1.js
mod2.val=NaN
```

ES6 Modules

More recent versions of Node.js support also ES6 modules

```
// include "type": "module" in 'package.json' to use the '.js' extension
// file points.mjs
class Point {
    constructor(x, y) { this.x = x; this.y = y; }
    move(dx, dy) { this.x += dx; this.y += dy; }
}
const origin = new Point(0, 0);
export { Point, origin };
```

Example: how to import definitions

```
// file use_points.mjs
import { Point, origin } from './points.mjs';
let p = new Point(1, 2);
p.move(1, 0);
console.log(p, origin);
```

Output: Point { x: 2, y: 2 } Point { x: 0, y: 0 }

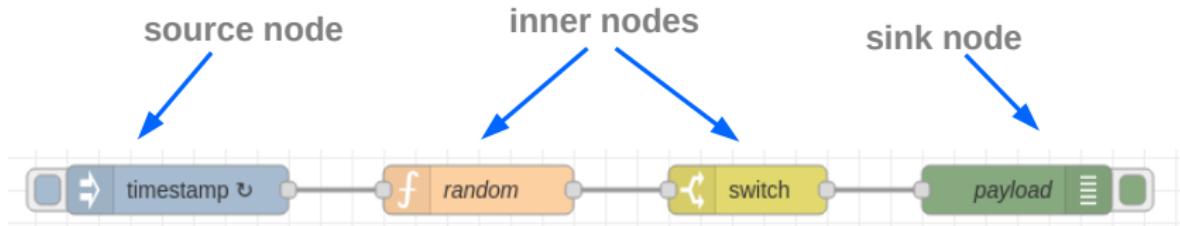
Node-RED



What is Node-RED?

- a programming tool for **event-driven applications**
- based on
 - Flow-based programming
 - Low-code programming
- particularly suitable for **wiring** the Internet of Things
- implementation based on **Node.js**
 - an execution platform which is a **Node.js web application**
 - the web application provides also a browser-based flow editor
 - runs on **Raspberry Pi, Android** (via Termux), interacts with **Arduino**
- developed by **IBM's Emerging Technology Services team**
 - Node-RED became **open source** in September 2013
 - it is now under the **OpenJS Foundation**
- documentation and tutorial available at <https://nodered.org/>

Node-RED



Flow-based programming

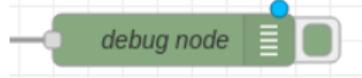
- programs are collections of **flows**
- flows are sequences of interconnected **nodes** where data flow and are processed
- **nodes** are the basic blocks of flows and perform simple operations on flowing data:
 - they have zero or one input, zero or more outputs
 - the initial node of the flow has no input and is a **source** of data received from external components or generated by the node itself
 - the final node of the flow has no output and is a **sink** of data sent to other components or displayed by the node itself

Node-RED

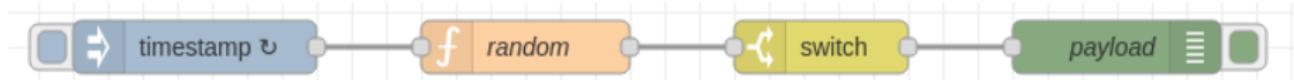
Example of initial nodes



Example of final nodes



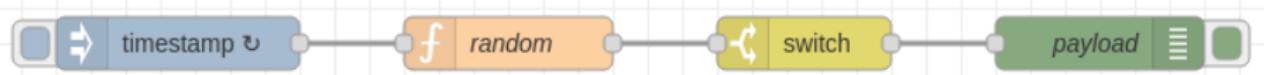
Node-RED



Low-code programming

- code has an intuitive **visual view**
- flows can be saved into **JSON files**
- flows and nodes are created and composed with a **graphical editor**
- nodes are configured with **pop-up menus**
- some general node requires insertion of specific **JavaScript/Node.js code**

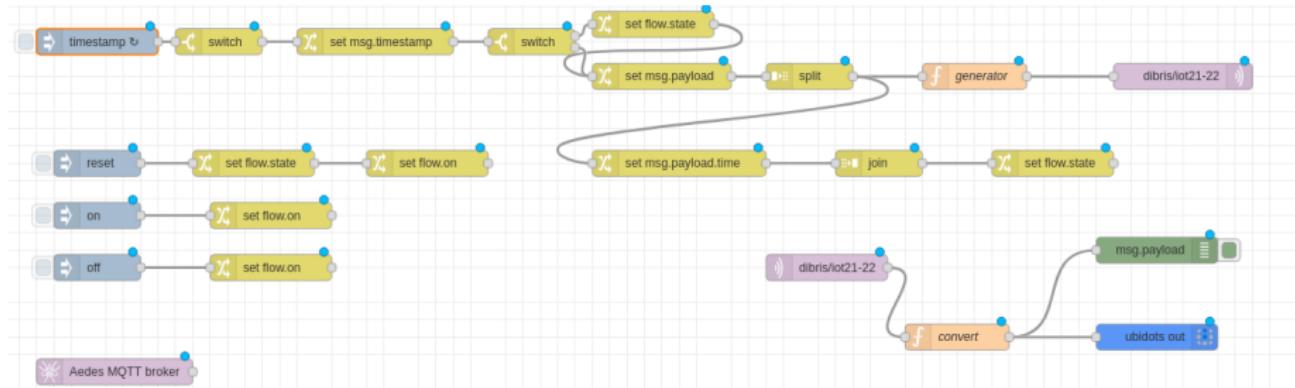
Node-RED



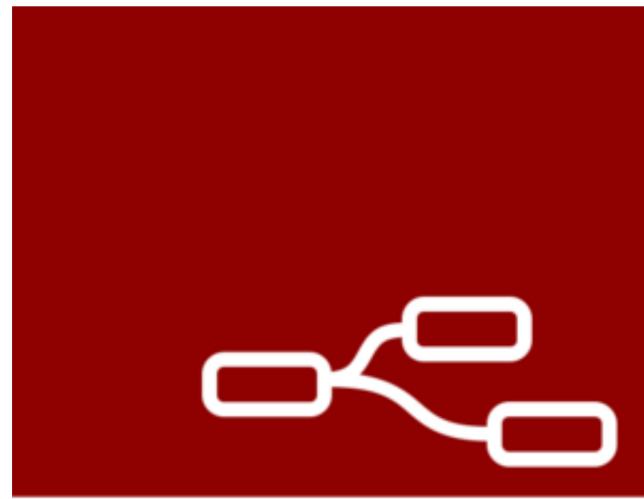
Overview of the runtime model

- a node can perform asynchronous operations (typically I/O)
- it triggers the nodes connected to its outputs by sending messages with the variable `msg` containing an object with property `payload` (and possibly others)
- messages between nodes are passed **asynchronously**
- nodes can access the following predefined variables:
 - `node`: the object representing the node itself
 - `msg`: the received and sent messages
 - `flow`: the keys which can be accessed (with methods `get`, `set` and `keys`) by all nodes on the same flow (or tab in the editor)
 - `global`: the keys which can be accessed (with methods `get`, `set` and `keys`) by all nodes

Node-RED demos



Node-RED: a tool for rapid IoT development



Node-RED

Main features Node-RED

flow-based

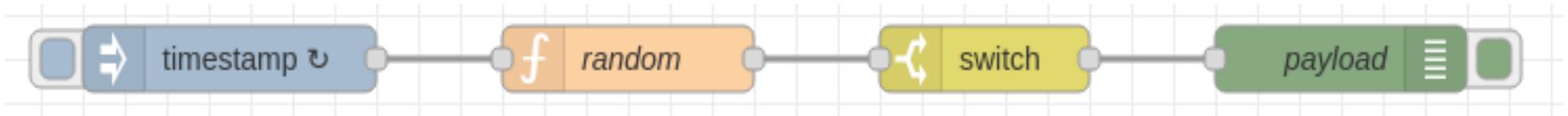
low-code

- a tool for event-driven programming
- based on
 - low-code programming
 - flow-based programming
- open source, based on Node.js
- libraries for IoT programming

open-source
Node.js

event-driven,
IoT

Flow-based programming



a Node-RED program is a collection of *flows*

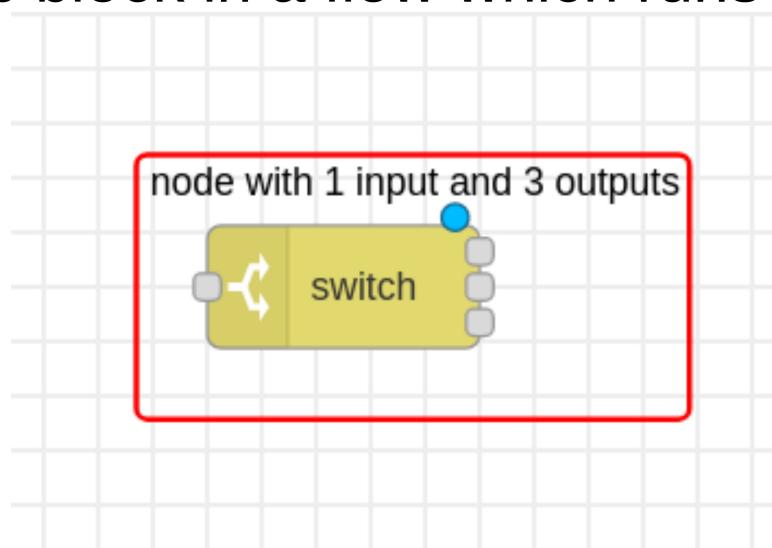
Terminology

a flow is a sequence of connected computational nodes
where data flow and are processed

Nodes

Terninology

a node is a basic block in a flow which runs simple tasks on the flowing data

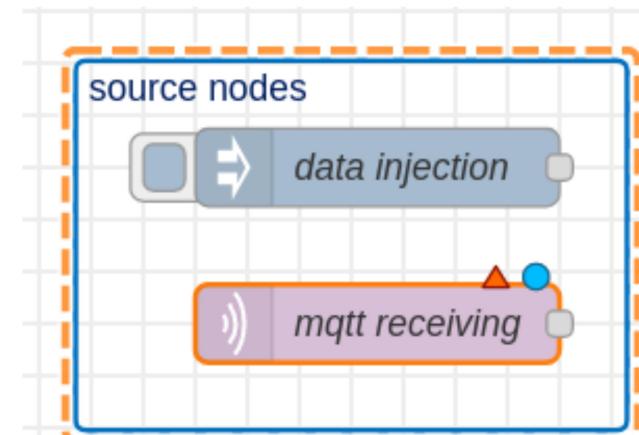
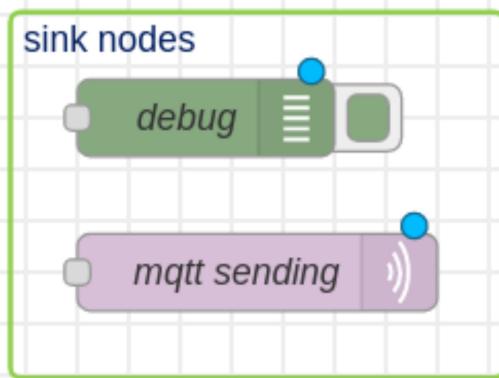


- a node has at most one input and zero or more outputs
- a node with no input is called a *source*
- a node with no outputs is called a *sink*
- all other nodes are called *inner*

Examples of nodes

source nodes

- injection of data in the flow
- receiving data over the MQTT protocol

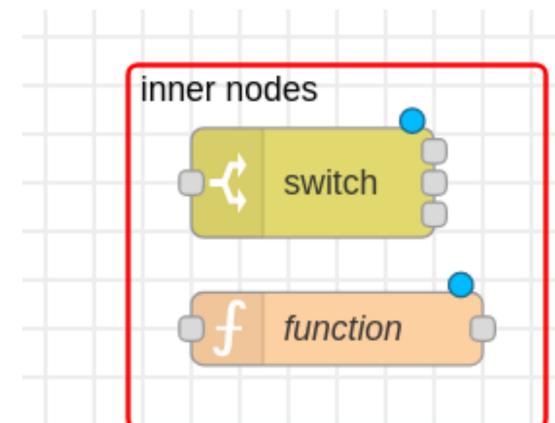


sink nodes

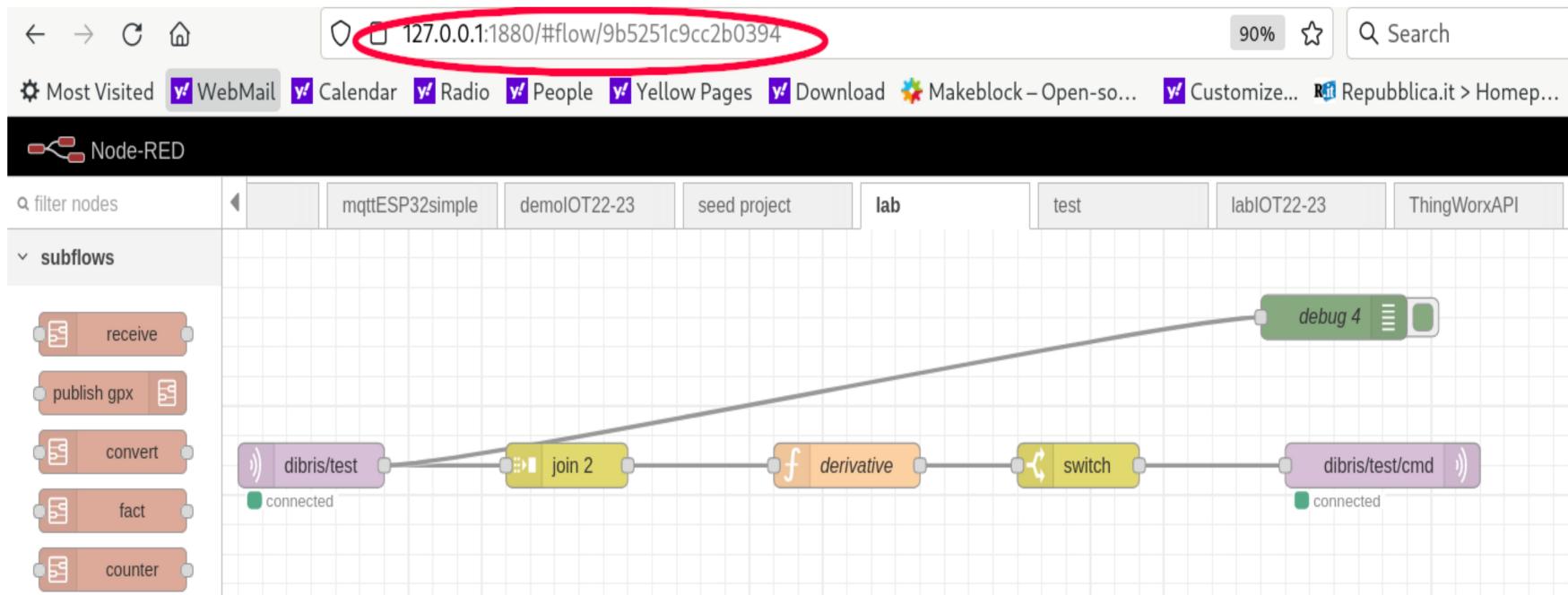
- debugging for displaying results
- sending data over the MQTT protocol

inner nodes

- switch: connects the input to zero or more outputs
- function: user-defined JavaScript code

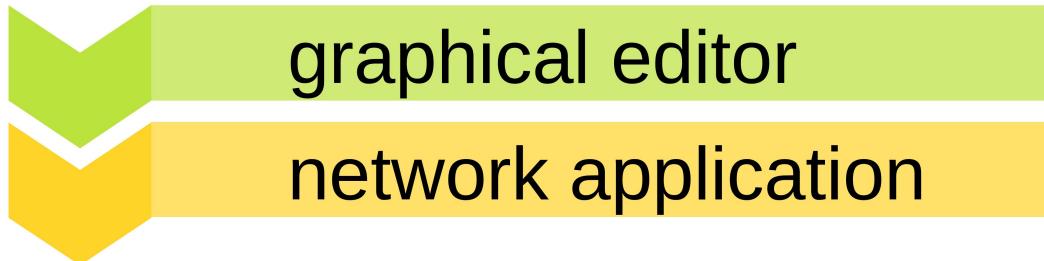


Some operational details



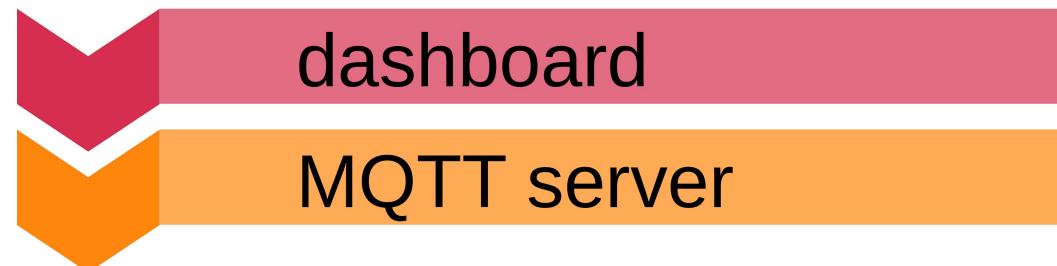
- any web browser can be used to develop and run a Node-RED program
- indeed, Node-RED allows the execution of a server with several main functionalities

Some operational details



Main features:

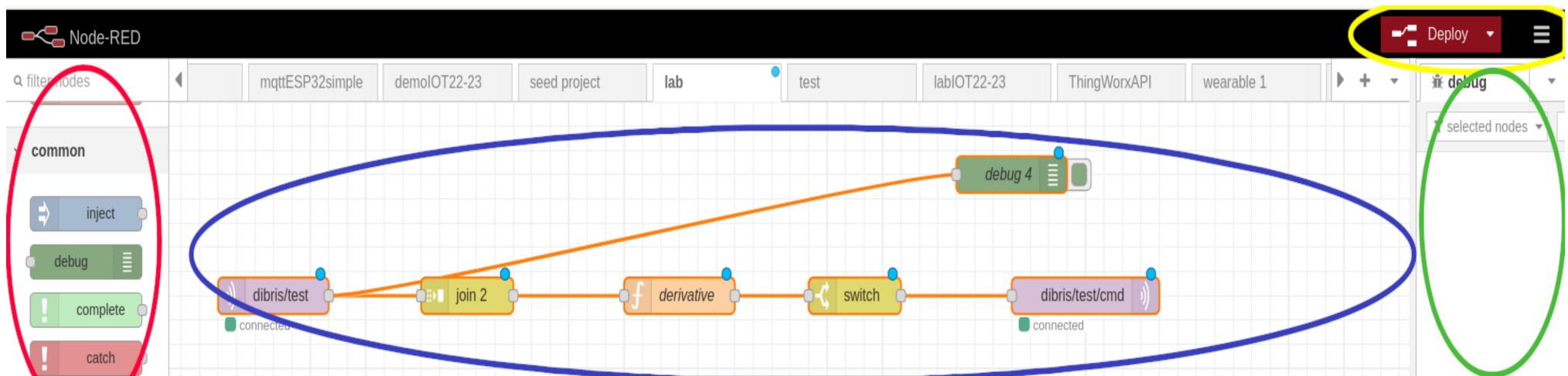
- graphical editor which allows the user to define the program flows by means of a web interface
- user-defined application able to interact with other network components over several protocols



Optional features:

- dashboard visualization
- MQTT server

Some operational details

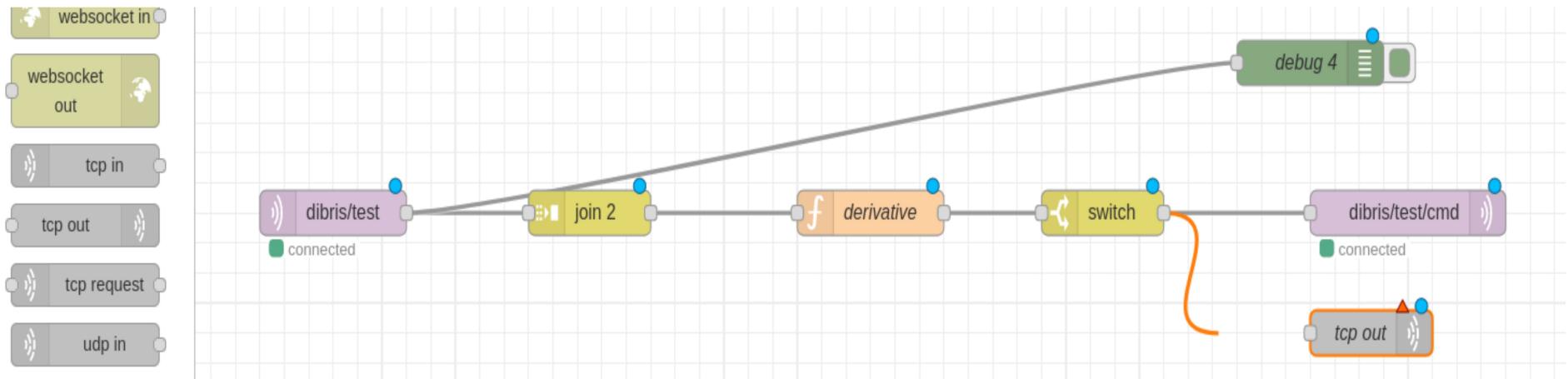


the graphical editor consists of four components:

- a header with the main menu and the “Deploy” button
- a palette on the left, with all types of available nodes
- a central workspace to define the program flows
- a sidebar on the right with multiple functionalities

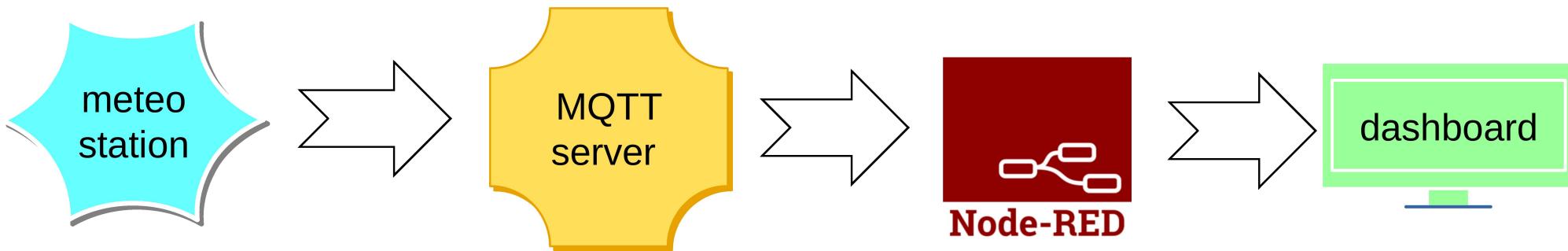
new types of nodes can be added to the palette by querying the available libraries on the web

Some operational details



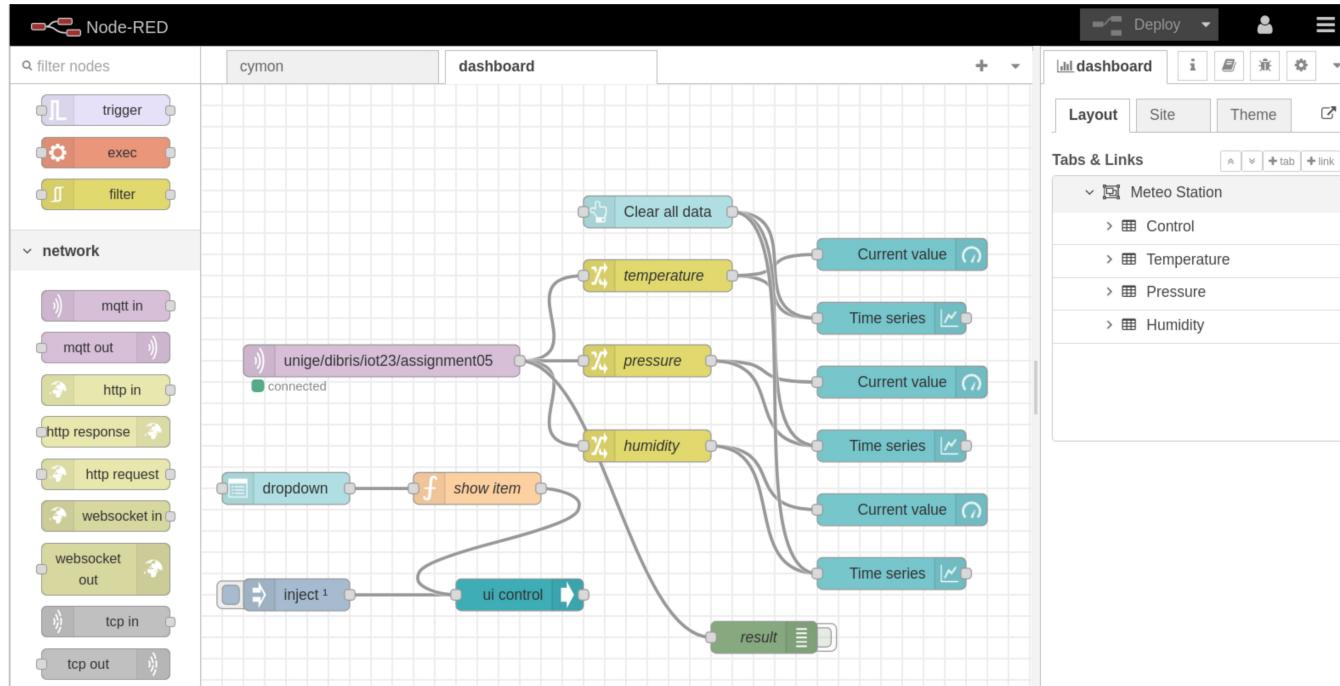
- new nodes can be added in a flow by dragging and dropping them in the workspace from the palette
- nodes are connected by holding down the left button of the mouse starting from an input or an output
- nodes can be configured with a double click of the left button of the mouse
- after all flows have been defined, the program can be run with the “Deploy” red button on the left top

Example of IoT application



- the application acquires data from a weather station equipped with temperature, humidity and pressure sensors
- data are sent through an MQTT server
- the Node-RED application subscribes to the MQTT server, elaborates the relevant data and displays them on the dashboard

Application flows

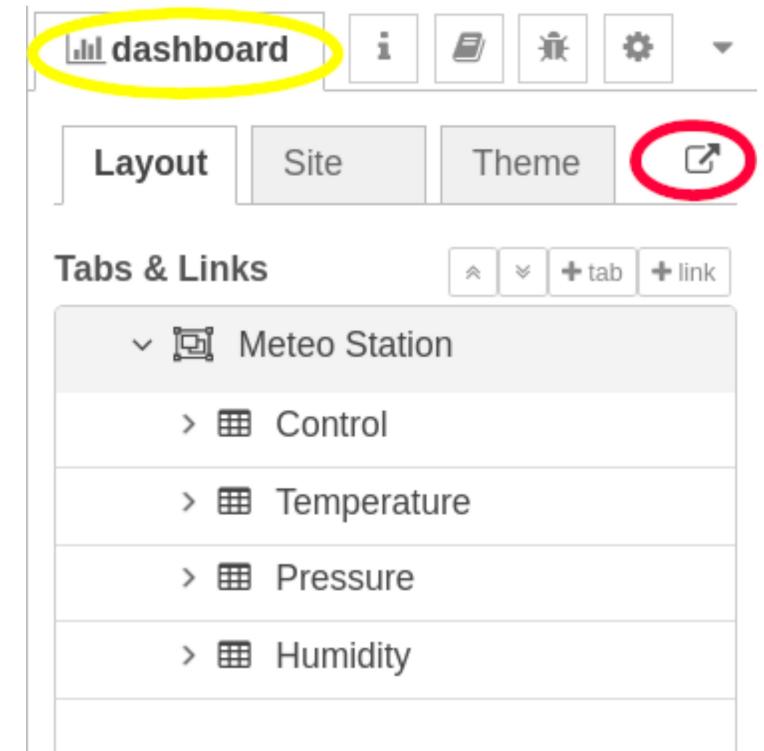


two main flows

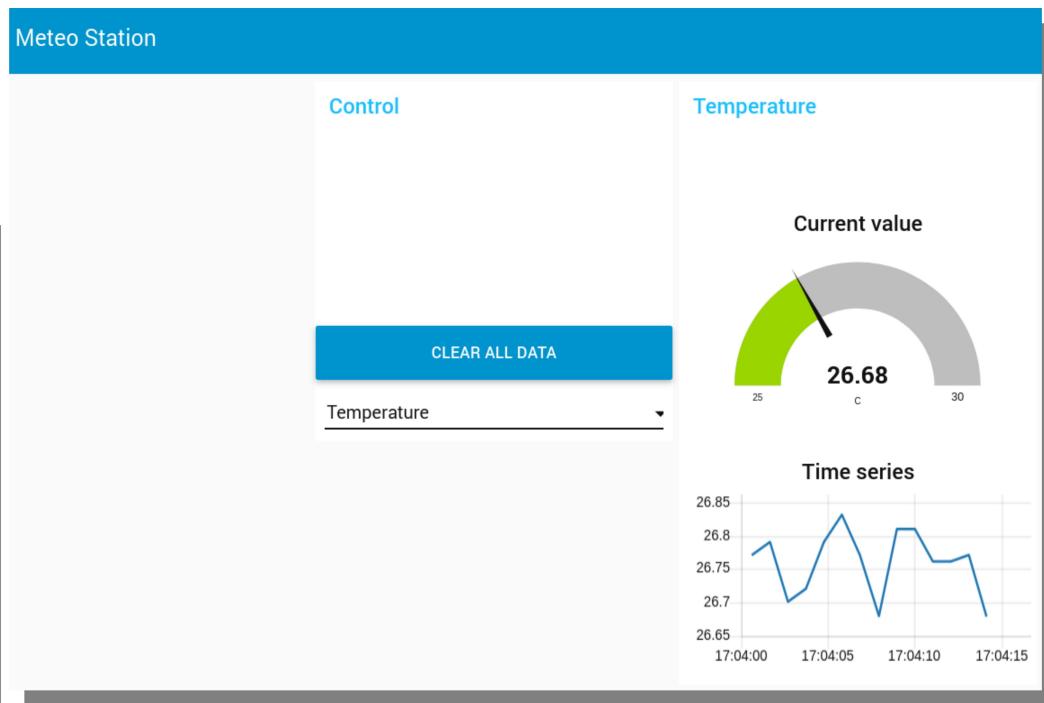
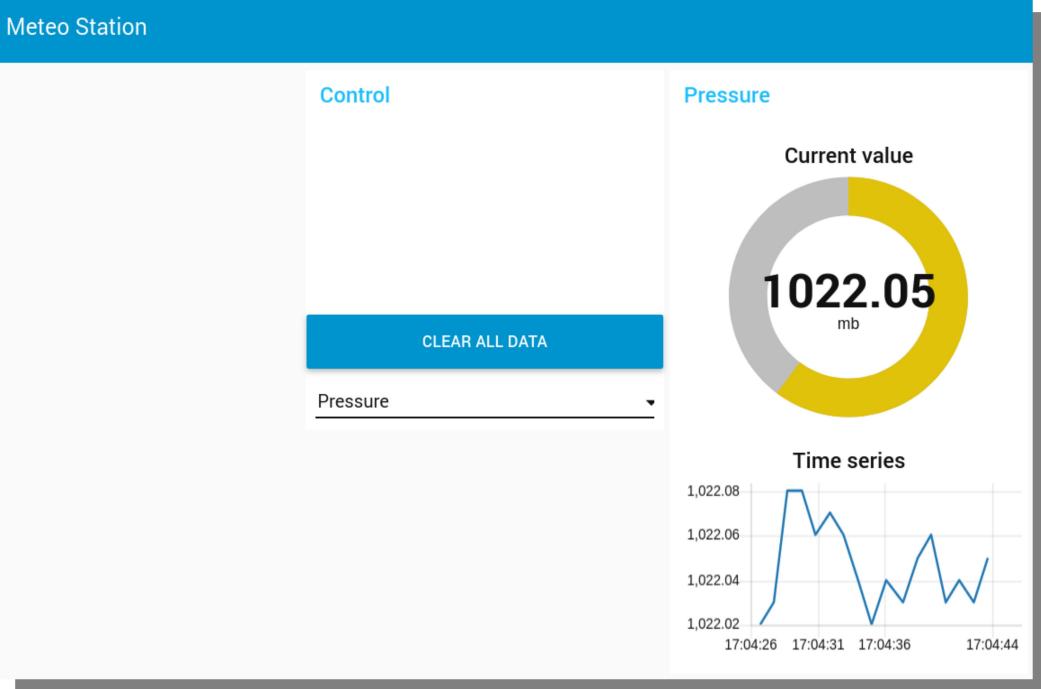
- data acquisition, elaboration and visualization with charts and gauges
- menu handling to allow data visualization for specific types of sensor (temperature, humidity and pressure)

Dashboard definition

- the “dashboard” tab in the sidebar allows the configuration of the properties of the web page where the application dashboard is displayed
- in the “dashboard” tab the right top arrow points to the link of the application dashboard



Example of visualization



the “clear all data” button in the dashboard deletes all data displayed in the chart

what have we learned?



- the main features and functionalities of Node-RED
- what is low-code and flow-based programming
- the use of the graphical editor of Node-RED
- how to develop with Node-RED a simple application for data acquisition, elaboration, and visualization for IoT devices
- how to display real-time data with a dashboard on the web

Device and Microcontroller Programming

Slides Giorgio Delzanno and Vittorio Sanguineti

Micro-Controller (MCU)

- A small computer on a single integrated circuit
 - One or more central processing units (CPUs)
 - Memory
 - Programmable input/output peripherals
- Data and programs are stored separately (Harvard architecture)
 - Program memory (non-volatile)
 - Data memory (volatile)
- In contrast to the microprocessors used in personal computers or other general purpose applications consisting of various discrete chips, MCUs are designed for **embedded applications**

Embedded Systems

- An *embedded system* is a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a specific function
- It is typically a component of a larger device or system
- In a well-designed embedded system, processor and software could go completely unnoticed by users of the device (eg. microwave oven, alarm clock, smartphone, **wearable devices**...)

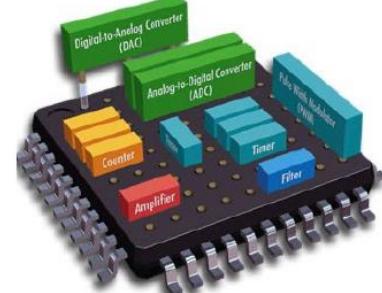
Design of an embedded system

- To design an embedded system, need to
 1. Design electronic circuit (microcontroller plus additional components, typically sensors and actuators)
 2. Develop and test program(s) to be run onboard

Software development in embedded systems

- **Problem:** don't have keyboard, monitor, operating system...
- **Solution:** develop software program on a host PC (with keyboard, monitor, graphic editor and all the like); then translate (compile) into binary code suitable for the embedded system
- **Cross-compiler:** a compiler capable of creating executable code for a platform other than the one on which the compiler is running
 - Separates build environment from target environment

Cross-compiler



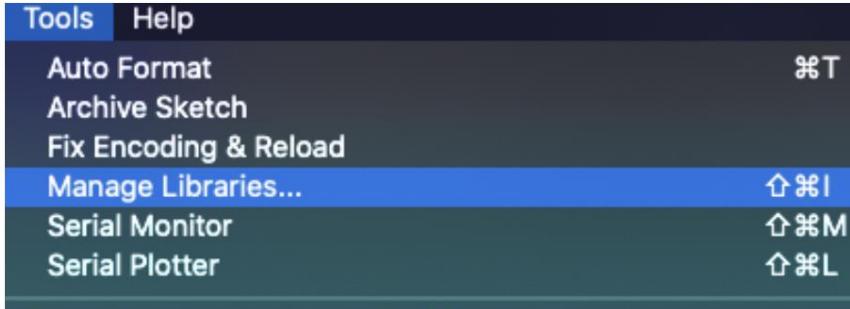
PC (Host)

- Write program (typically in high-level language, like C/C++,...)
- Cross-compile program
- Link compiled code with existing libraries and generate executable program
- Upload executable into embedded system

Embedded system (Target)

- Run the executable program (forever)

Libraries Management



The screenshot shows the Arduino IDE's Tools menu open. The "Manage Libraries..." option is highlighted with a blue background and white text. Other options like Auto Format, Archive Sketch, Fix Encoding & Reload, Serial Monitor, and Serial Plotter are also listed.

Below the menu, the main workspace displays several library examples:

- Arduino Cloud Provider Examples**: by Arduino. Examples of how to connect various Arduino boards to cloud providers. Version 1.0.0.
- Arduino Low Power**: by Arduino. Version 1.2.2 INSTALLED. Power save primitives features for SAMD and nRF52 32bit boards. With this library you can manage the low power states of your board.
- Arduino SigFox for MKRFox1200**: by Arduino. Helper library for MKRFox1200 board and ATAB8520E Sigfox module. This library allows some high level operations on Sigfox ease integration with existing projects. Version 1.0.0.

A progress bar at the bottom indicates "Updating list of installed libraries".

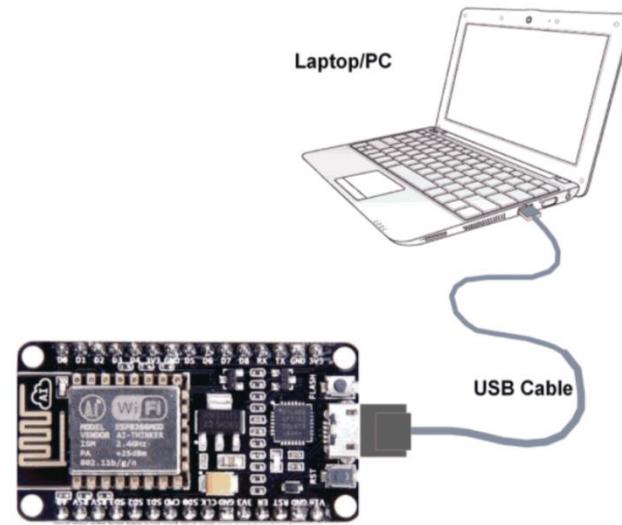
Serial Monitor



The Serial Monitor window shows the following JSON data being printed:

```
02:50:58.709 ->      ,
02:50:58.709 ->      {
02:50:58.709 ->          "items": [
02:50:58.744 ->              "Why <em>WonderWidgets</em> are great",
02:50:58.744 ->              "Who <em>buys</em> WonderWidgets"
02:50:58.744 ->          ],
02:50:58.744 ->          "title": "Overview",
02:50:58.744 ->          "type": "all"
02:50:58.744 ->      }
02:50:58.744 ->      ],
02:50:58.744 ->          "title": "Sample Slide Show"
02:50:58.744 ->      }
02:50:58.744 ->
```

At the bottom of the window, there are checkboxes for "Autoscroll" and "Show timestamp", and a dropdown for "Baud rate" set to "115200 baud".



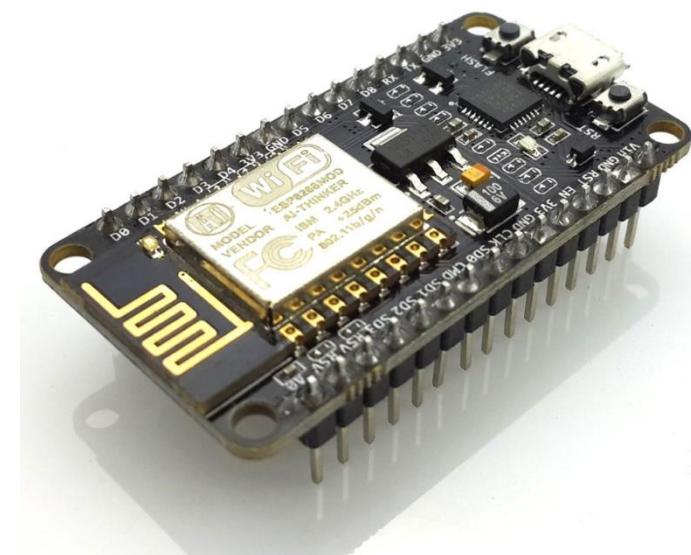
The screenshot shows the Arduino IDE's Tools menu open. The "Serial Monitor" option is highlighted with a blue background and white text. Other options like Auto Format, Archive Sketch, Fix Encoding & Reload, Manage Libraries..., and Serial Plotter are also listed.

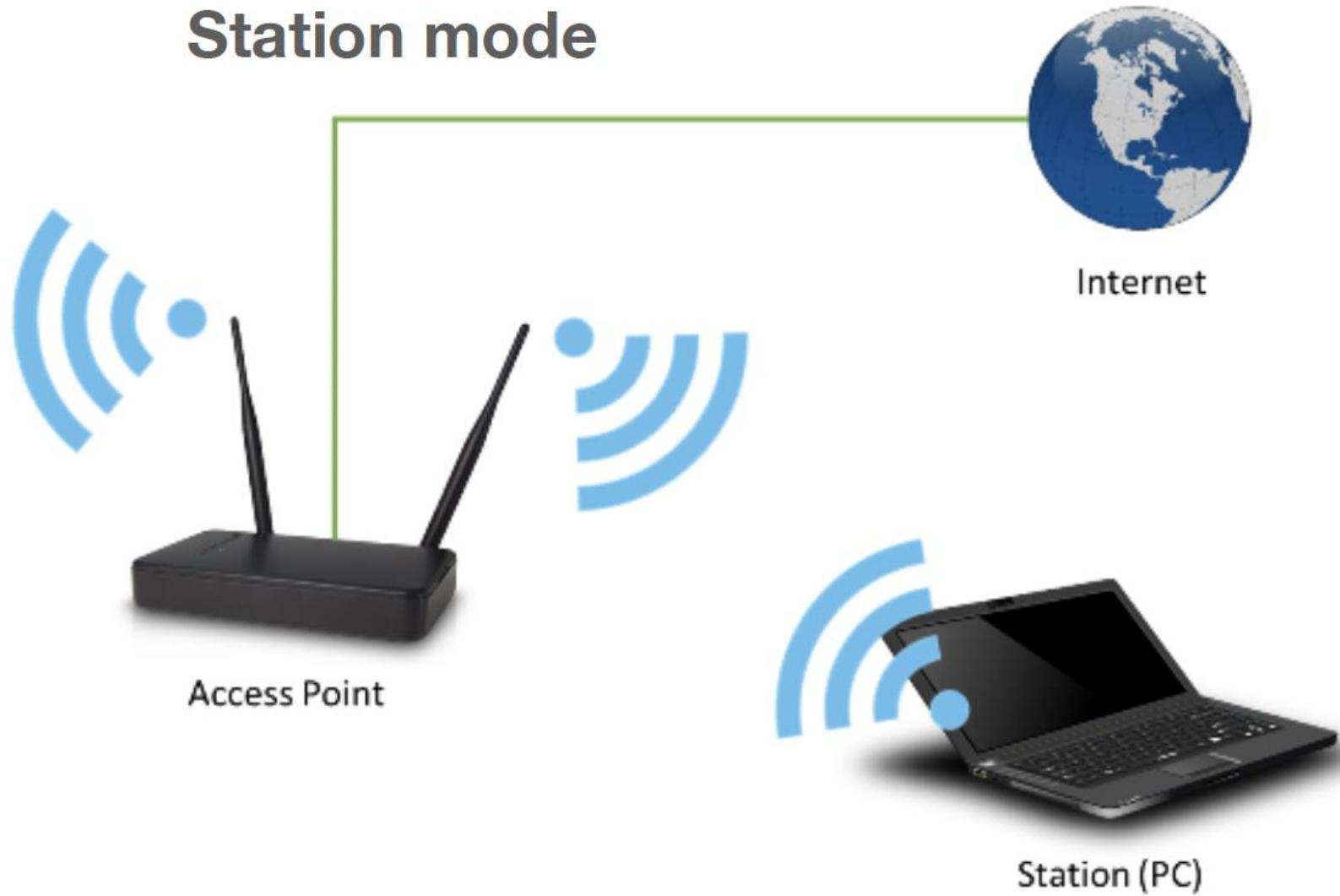
Below the menu, the main workspace displays the WiFi101 / WiFiINA Firmware Updater settings:

- Board: "NodeMCU 1.0 (ESP-12E Module)"
- Builin Led: "2"
- Upload Speed: "115200"
- CPU Frequency: "160 MHz"
- Flash Size: "4MB (FS:none OTA:~1019KB)"
- Debug port: "Disabled"
- Debug Level: "None"
- IwIP Variant: "v2 Lower Memory"
- VTables: "Flash"
- Exceptions: "Disabled (new can abort)"
- Erase Flash: "Only Sketch"
- SSL Support: "All SSL ciphers (most compatible)"
- Port: "/dev/cu.SLAB_USBtoUART"
- Get Board Info
- Programmer
- Burn Bootloader

At the bottom, there are buttons for "Newline", "115200 baud", and "Clear output".

Examples of Networking (e.g. ESP8286 equipped with WiFi interface)





Soft AP mode station mode



Station
(mobile phone)



Soft Access Point
(ESP8266)

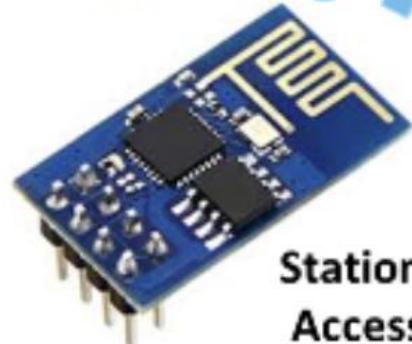


Station (PC)

Node of a mesh protocol



Station (PC)



Station + Soft
Access Point
(ESP8266)



Access Point



Internet

Coding for embedded systems

- Issues:
 - small program memory size (need efficiency)
 - must access hardware (peripherals)
- Other desirable features:
 - modularity, code reuse, data abstraction (system peripherals as classes)
- Solution: use high-level languages (C/C++, Python, etc)

Only one program

- A typical embedded system only runs one program, which is launched at power-up and runs forever:

```
main() {  
    // code which must only be run once  
  
    while(true){  
        // code which must be run continuously  
    }  
}
```

Arduino platform

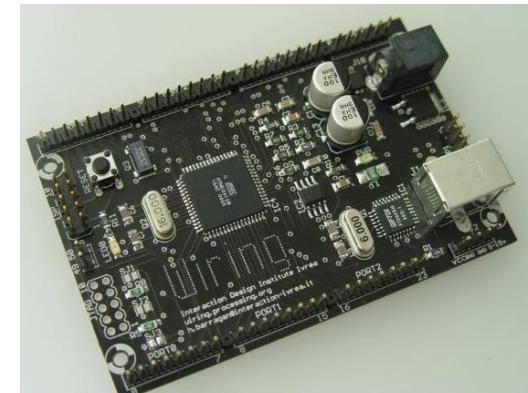
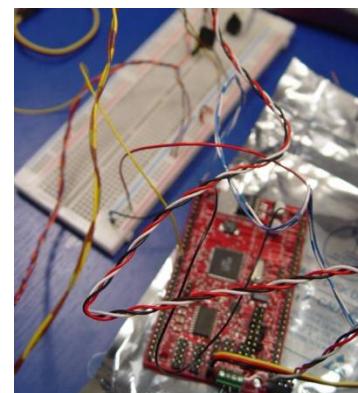
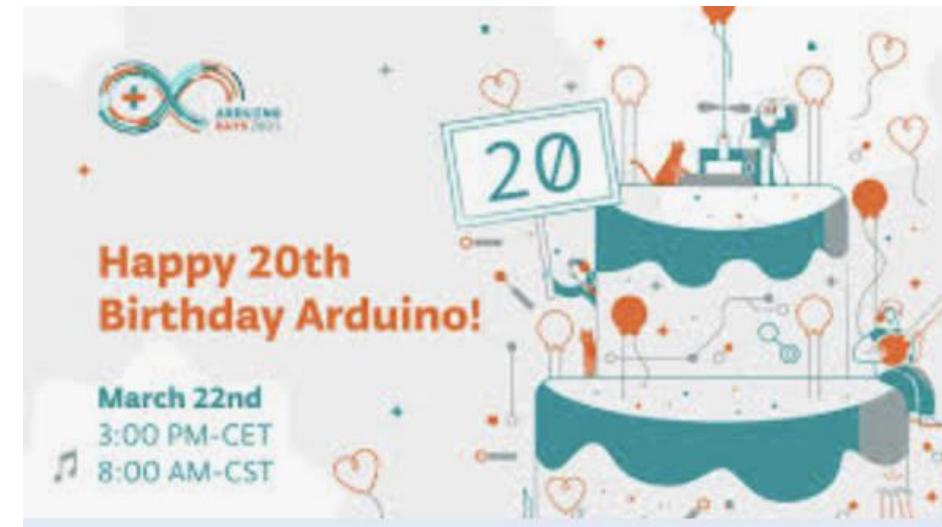


- Open-source electronics prototyping platform based on flexible, easy-to-use hardware and software
- A family of software-compatible boards plus a large variety of compatible hardware modules ('shields') equipped with a variety of sensors and actuators and/or specific functions
- Allows to develop prototypes of complex electronic devices by combining pre-existing modules
- All electronic circuits are open-source, hence a prototype can easily evolve into a special purpose device

20 years of Arduino's history

<https://arduinohistory.github.io/>

- The project began as a tool for students at the Interaction Design Institute Ivrea, Italy,
- The goal was to provide a low-cost and easy way to create devices that interact with their environment using sensors and actuators.
- They started from a BASIC Stamp microcontroller at a cost of \$50
- The development platform Wiring was created in a project at IDII, under the supervision of Massimo Banzi and Casey Reas the co-creator of the Processing programming language
 - A PCB with an ATmega128 microcontroller, an IDE based on Processing and library functions to program the microcontroller.



20 years of Arduino's history

<https://arduinohistory.github.io/>



- In 2005, Massimo Banzi and his students extended Wiring by adding support for the cheaper ATmega8 microcontroller.
The new project, forked from Wiring, was called *Arduino* (a bar in Ivrea)
- In 2008, the co-founders created Arduino LLC: manufacture and sale of the boards were done by external companies, Arduino LLC would get a royalty from them)
- In 2018, Arduino announced its new open source command line tool ([arduino-cli](#)), a replacement of the IDE to program the boards from a shell.
- In 2019, Arduino announced its IoT Cloud service
- In 2025, the Arduino community included about 30 million active users based on the IDE downloads.

Key Features

- **Hardware:**
 - Standard form factor that breaks the functions of the microcontroller into a more accessible package
 - **Analog and Digital I/O.** Read analog and digital input and turn it into an output (activating motor, on/off a Led, connect to the cloud..)
 - **USB port.** Used for communication with PC and power supply
- **Software:**
 - Runs a single program (a ‘sketch’)
 - Simplified version of C++
 - Arduino integrated development environment (IDE) running on a PC



Arduino Nano 33 IoT



Arduino Nano
RP2040 Connect



Arduino Nano ESP32



Arduino Nano 33
BLE Sense



Arduino MKR 1000 WiFi



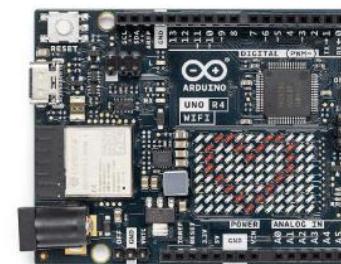
Arduino MKR WiFi 1010



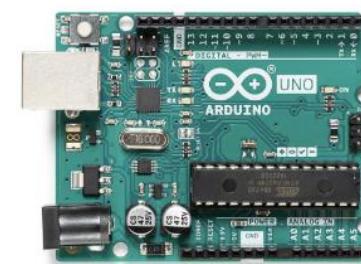
Arduino MKR FOX 1200



Arduino UNO R4
Minima



Arduino UNO R4
WiFi



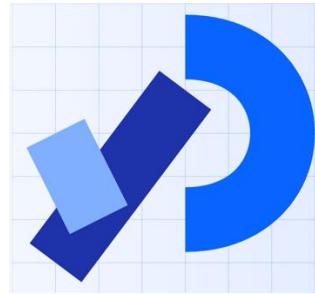
Arduino UNO R3

Arduino IDE 1.0

The screenshot shows the Arduino IDE 1.0 interface with the "Blink" example sketch open. The code is as follows:

```
/*  
 *  
 * Blink  
 *  
 * Turns on an LED on for one second, then off for one second, repeatedly.  
 *  
 * This example code is in the public domain.  
 */  
  
void setup() {  
    // initialize the digital pin as an output.  
    // Pin 13 has an LED connected on most Arduino boards:  
    pinMode(13, OUTPUT);  
}  
  
void loop() {  
    digitalWrite(13, HIGH); // set the LED on  
    delay(1000); // wait for a second  
    digitalWrite(13, LOW); // set the LED off  
    delay(1000); // wait for a second  
}
```

At the bottom, it says "1" and "Arduino Uno on /dev/tty.usbmodem621".



Processing IDE

The screenshot shows the Processing IDE interface with a sketch titled "sketch 250413a". The code is as follows:

```
1  /**  
2   * Rotate Push Pop.  
3   *  
4   * The push() and pop() functions allow for more control over transformations.  
5   * The push function saves the current coordinate system to the stack  
6   * and pop() restores the prior coordinate system.  
7   */  
8  
9  float a; // Angle of rotation  
10 float offset = PI/24.0; // Angle offset between boxes  
11 int num = 12; // Number of boxes  
12  
13 void setup() {  
14     size(640, 360, P3D);  
15     noStroke();  
16 }  
17  
18 void draw() {  
19     lights();  
20  
21     background(0, 0, 26);  
22     translate(width/2, height/2);  
23 }
```

At the bottom, there are tabs for "Console" and "Error".

Arduino IDE 2.0

Arduino UNO R4 WiFi

LIBRARY MANAGER

Filter your search...

Type: All

Topic: All

AIPlc_Opta by Arduino

Arduino IDE PLC runtime library for Arduino Opta This is the runtime library and plugins fo...

More info

1.2.0 **INSTALL**

AIPlc_PMC by Arduino

Arduino IDE PLC runtime library for Arduino Portenta Machine Control This is the runtime...

More info

1.0.6 **INSTALL**

Arduino Cloud Provider Examples b...

Examples of how to connect various Arduino boards to cloud providers

More info

1.2.1 **INSTALL**

schema.ino BotBidArduino.ino Services.h arduino_secrets.h

```
1 #include "arduino_secrets.h"
2 /Users/giorgiodelzanno/Documents/_PROGETTI/_BotBid/BotBidArduino/
3 schema.ino
4
5 INIZIALIZZAZIONE DEI PARAMETRI DI CONNESSIONE E CALIBRAZIONE DEI SENSORI
6 INIZIALIZZAZIONE DELLE VARIABILI PER OSSERVARE DATI
7
8 void setup() {
9     Inizializzazione del monitor seriale (per debugging)
10    Inizializzazione del sensore ambientale
11    Connessione al WiFi
12    Connessione al server NTP per timestamp dalla rete Internet
13 }
14
15 void loop() {
16     Tentativo di riconnessione al WiFi in caso di disconnessione
17     Lettura del voltaggio corrente e calcolo del PH
18     Lettura dei dati del sensore ambientale
19     Stampa dei dati sul monitor seriale (per debugging)
20     Lettura ora corrente da server RTC
21     Invio dei dati al server BotBid se è passato un minuto dall'ultimo update e
22     i dati sono stati raccolti correttamente
23 }
24
25 /* -----
26 // funzione per mandare i dati raccolti dai sensori al server
27 void sendData() {
28     Tentativo di connessione al server cloud Firebase
29     https://bot-bid-default.firebaseio.com/.json
30
31     Preparazione della richiesta a HTTP PUT con credenziali user ed esperimento
32 }
```

Output

Ln 6, Col 52 Arduino UNO R4 WiFi on /dev/cu.wlan-debug



Arduino Nano 33 IoT

Arduino Nano
RP2040 Connect

Arduino Nano ESP32

Arduino Nano 33
BLE Sense

Arduino Cloud

The screenshot shows the Arduino Cloud interface for a sketch named "BotBidArduino.ino". The code includes #include statements for "aux.h", "Services.h", and "DFRobot_EnvironmentalSensor.h", and defines MODESWITCH as either UART or I2C. It also initializes SoftwareSerial objects for both modes. The interface includes a sidebar with icons for file operations like "NUOVO", a "Consolle" (Console) section at the bottom, and a top bar with a "SELEZIONA DISPOSITIVO" button.

```
#include "aux.h"
#include "Services.h"
#include "DFRobot_EnvironmentalSensor.h"

// connessione al modulo ambientale di tipo I2C
#define MODESWITCH /*UART: 1*/ 0 /*I2C: 0*/

#if MODESWITCH
#if defined(ARDUINO_AVR_UNO) || defined(ESP8266)
SoftwareSerial mySerial(/*rx =*/4, /*tx =*/5);
DFRobot_EnvironmentalSensor environment(/*addr =*/SEN050X_DEFAULT_DEVICE_ADDRESS, /*s =*/&mySerial);
#else
DFRobot_EnvironmentalSensor environment(/*addr =*/SEN050X_DEFAULT_DEVICE_ADDRESS, /*s =*/&Serial1);
#endif
#endif
```

The advertisement features the Arduino Cloud logo and the tagline "Bring your IoT projects to life quickly". Below it is the text "Your next exciting journey to build, control and monitor your connected projects".

ARDUINO CLOUD

Bring your IoT projects to life quickly

Your next exciting journey to build, control and monitor your connected projects

The screenshot shows the Arduino Cloud interface for managing devices. On the left, a sidebar lists "Casa", "Schizzi", and "Dispositivi" (Devices), with "Dispositivi" currently selected. The main area displays a table of devices:

Nome del dispositivo	Stato	Tipo	Cosa associata	Modulo di connettività
BotBid	OFFLINE	Arduino UNO R4 WiFi	-	0.4.1
MKR1000	OFFLINE	Arduino MKR1000	Senza titolo	19.6.1

VSE PlatformIO IDE

The screenshot shows the VS Code Marketplace interface. On the left, there's a sidebar with various icons and a list of extensions. The main area displays the details for the "PlatformIO IDE" extension.

Extension Details:

- Name:** PlatformIO IDE
- Provider:** PlatformIO (platformio.org)
- Version:** 5.359.196
- Description:** Your Gateway to Embedded Software Development
- Status:** Disinstalla (Uninstall) | Aggiornamento automatico (Automatic update)

PlatformIO IDE for VSCode

Installazione (Installation):

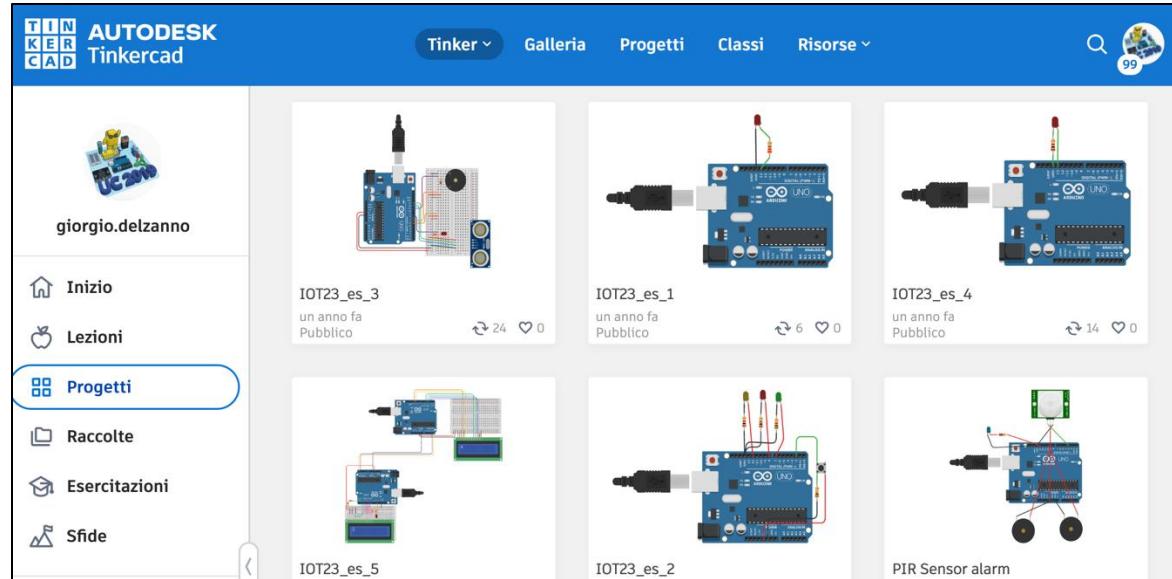
- Identifier: platformio.platformio-ide
- Version: 3.3.4
- Last updated: 2025-04-13, 23:49:21
- Dimensions: 6.03 MB

Marketplace (Marketplace):

- Published: 2017-05-28, 00:02:51

Arduino Simulators

Autodesk Tinkercad



Wokwi

The screenshot shows the Wokwi web interface. At the top right are "Docs" and a user icon. Below is a search bar and a "NEW PROJECT" button. The main area is divided into two sections: "YOUR PROJECTS" and "YOUR LIKES".

YOUR PROJECTS

- mqtt_esp32_prova** (5 months ago)
Code:

```
// From https://randomnerdtutorials.com/esp32-mqtt-publish-subscribe-arduino/  
  
#include <WiFi.h>  
#include <PubSubClient.h>  
  
const char* ssid = "Wokwi-GUEST";  
const char* password = "";  
  
const char* mqttServer = "test.mosquitto.org";  
const int LED = 14;  
  
int status = WL_IDLE_STATUS;  
WiFiClient client;  
  
void setup() {  
    Serial.print("Connecting to WiFi...");  
}
```
- mqttESP32_example** (5 months ago)
Code:

```
void setup() {  
    Serial.print("Connecting to WiFi...");  
}
```

YOUR LIKES

- PIR Sensor alarm**

Microsoft Code Creator

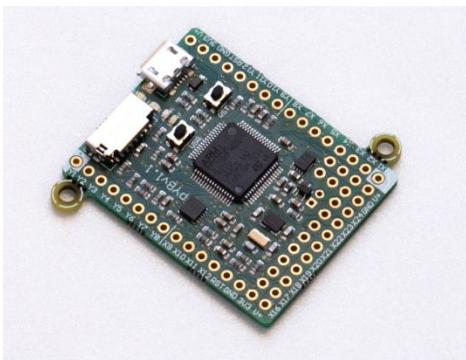
The screenshot shows the Microsoft Code Creator interface. It features a central workspace with a blue Arduino Uno board. To the right is a block-based programming palette with categories: LIGHT, INPUT, MUSIC, LOOPS, LOGIC, VARIABLES, MATH, EXTENSIONS, and ADVANCED. Two green blocks are placed in the workspace: "on start" and "forever". At the bottom are buttons for "Scaricamento" (Download), "Prova" (Test), and other navigation controls.

Other boards/IDE

micro:bit
let's code



pyboard
micro:python
ViperIDE

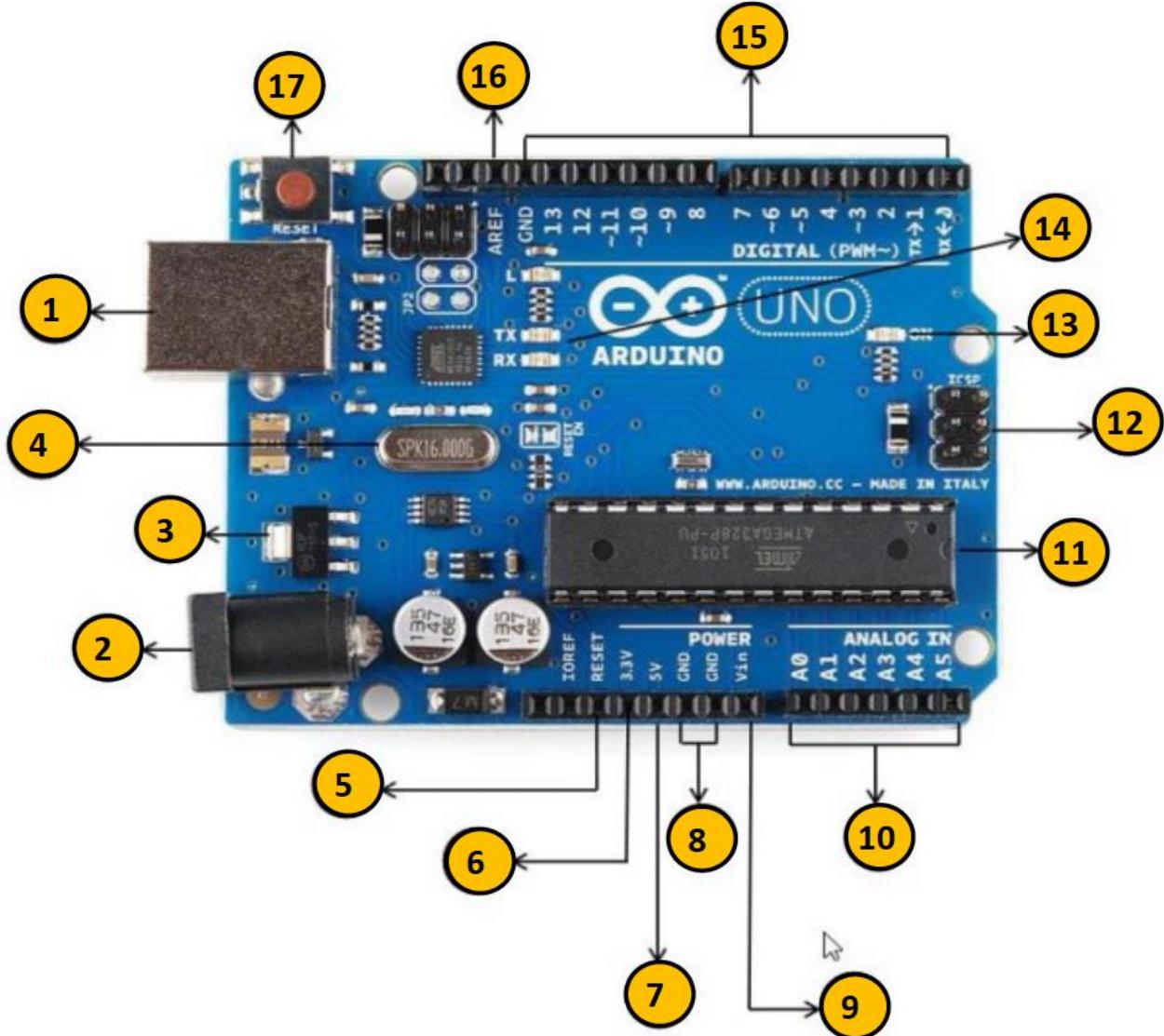


The Microsoft micro:bit web IDE interface. On the left, a preview window shows the micro:bit board with several red LEDs illuminated in a pattern. To the right is a block-based code editor with a sidebar containing categories like Fondamentali, Ingressi, Musica, etc. A script titled "all'avvio" is visible, containing blocks for setting variables n, r, and r2 to specific values.

ViperIDE MicroPython Web IDE interface. On the left, a file browser shows a single file named "test.py". The code editor window displays the following Python script:

```
1 # ViperIDE - MicroPython Web IDE
2 # Read more: https://github.com/vshymanskyy/ViperIDE
3
4 # Connect your device and start creating! 🚀🤖💡
5
6
7 # You can also open a virtual device and explore some examples:
8 # https://viper-ide.org?vm=1
9
```

Arduino UNO



1. Power USB
2. Power (Barrel Jack)
3. Voltage Regulator
4. Crystal Oscillator
5. Reset pin
6. Internal 3.3V source
7. Internal 5V source
8. GND
9. External power (V_{in})
10. Six analog inputs
11. Main microcontroller
12. In-Circuit Serial Programming (ICSP) pin
13. Power LED indicator
14. TX and RX LEDs
15. 14 Digital I/O pins
16. Analog Reference (AREF)
17. Reset button

Features

- **MCU:** ATmega328P
 - **Clock Speed:** 16 MHz
 - **Flash Memory:** 32 KB of which 0.5 KB used by bootloader
 - **SRAM** 2 KB (volatile)
 - **EEPROM** 1 KB (non volatile)
- **Operating Voltage:** 5V
- **Input Voltage** (recommended): 7-12V
- **Input Voltage** (limits): 6-20V
- **Digital I/O Pins:** 14 (of which 6 provide PWM output)
- **Analog Input Pins:** 6 (10 bits)

An Arduino sketch

- Arduino code is called a ‘sketch’
- C++-like native programming language
- General structure:

```
// #include directives
// global variables here

void setup() {
    // put your setup code here, to run once:

}

void loop() {
    // put your main code here, to run repeatedly:

}
```

Arduino sketch: global and local variables

Global variables are available to all functions in a program

Local variables are only visible to the function in which they are declared

In the Arduino environment (same as C++), any variable declared outside of a function (e.g. `setup()`, `loop()`, etc.), is a **global variable**

```
int Time=1000;  
int pinLed=13;
```

The Arduino UNO board has a LED which is connected to digital pin 13

```
void setup() {  
    //  
}
```

```
void loop() {  
    //  
}
```

Arduino sketch: setup() function

The `setup()` function will run only once, after each power-up or reset of the Arduino board

Use it to initialize variables, pin modes, start using libraries, etc

```
int Time=1000;  
int pinLed=13;
```

```
void setup() {  
    pinMode(pinLed,OUTPUT); //pin 13 initialization (LED)  
}
```

```
void loop() {  
    //  
}
```

Configures digital pin
#13 as output

Arduino sketch: loop() function

After creating a setup() function, which initializes and sets the initial values, the **loop()** function does precisely what its name suggests, and loops consecutively, allowing your program to change and respond. Use it to actively control the Arduino board.

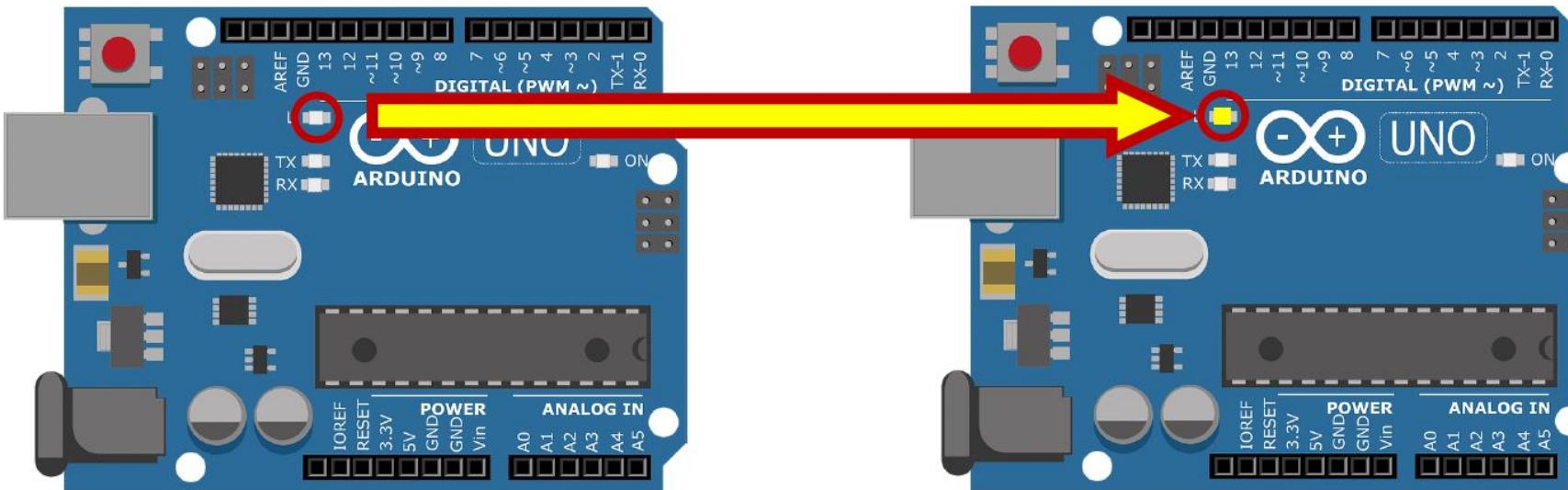
```
int Time=1000;
int pinLed=13;

void setup() {
    pinMode(pinLed,OUTPUT); //pin 13 initialization (LED)
}

void loop() {
    digitalWrite(pinLed,HIGH); //LED on
    delay(Time);
    digitalWrite(pinLed,LOW); //LED off
    delay(Time);
}
```

Turns LED on, waits 1 s,
turns off, waits 1 s, etc

Hello world (Arduino version)



Hello world (Arduino version)

```
int Time=1000;
int pinLed=13;

void setup() {
    pinMode(pinLed,OUTPUT); //pin 13 initialization (LED)
}

void loop() {
    digitalWrite(pinLed,HIGH); //LED on
    delay(Time);
    digitalWrite(pinLed,LOW); //LED off
    delay(Time);
}
```

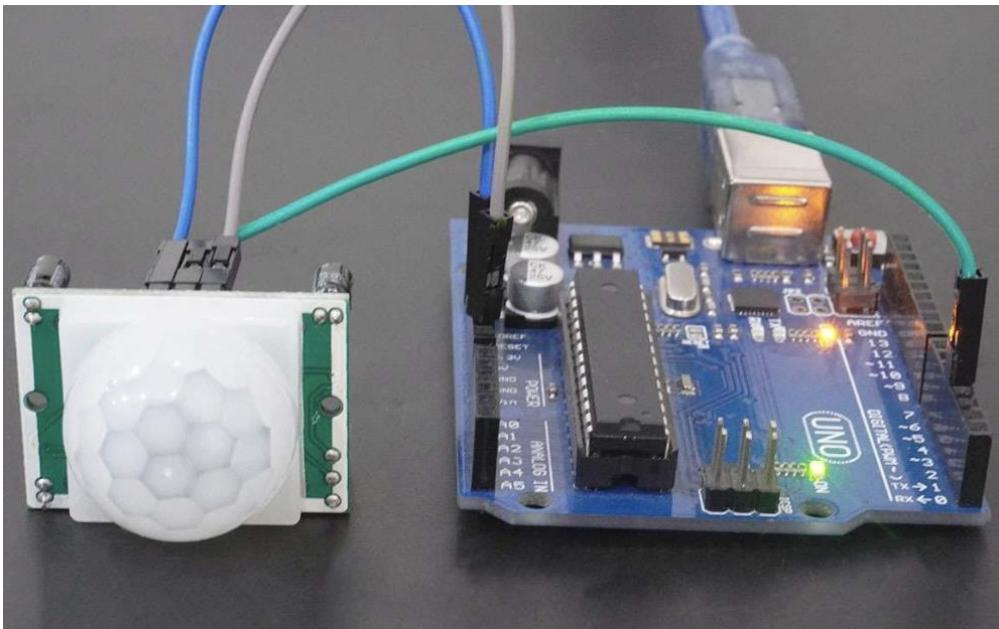
Arduino shields

- **Shields** are additional boards that can be plugged on top of the Arduino board, thus extending its capabilities.
- All shields follow the same philosophy as the base Arduino board: easy to mount, cheap to produce and freely available circuit diagram
- Many Arduino shields are stackable. You can connect many shields together to create a tower of Arduino modules

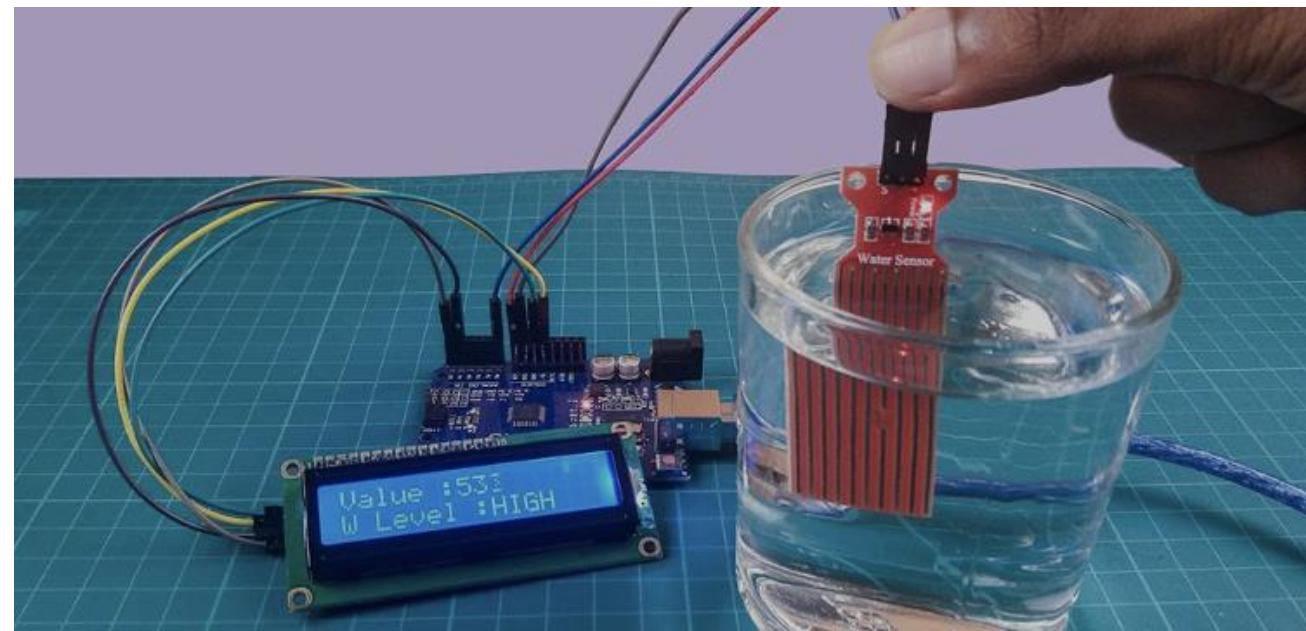
Shield form factor

- To be stackable, every Arduino shield must have the same form factor as the standard Arduino board
- When stacking shields, it is important to make sure they do not use overlapping pins
- Shields communicate with the Arduino board through
 - Analog inputs
 - Digital input/outputs, using a variety of communication modalities

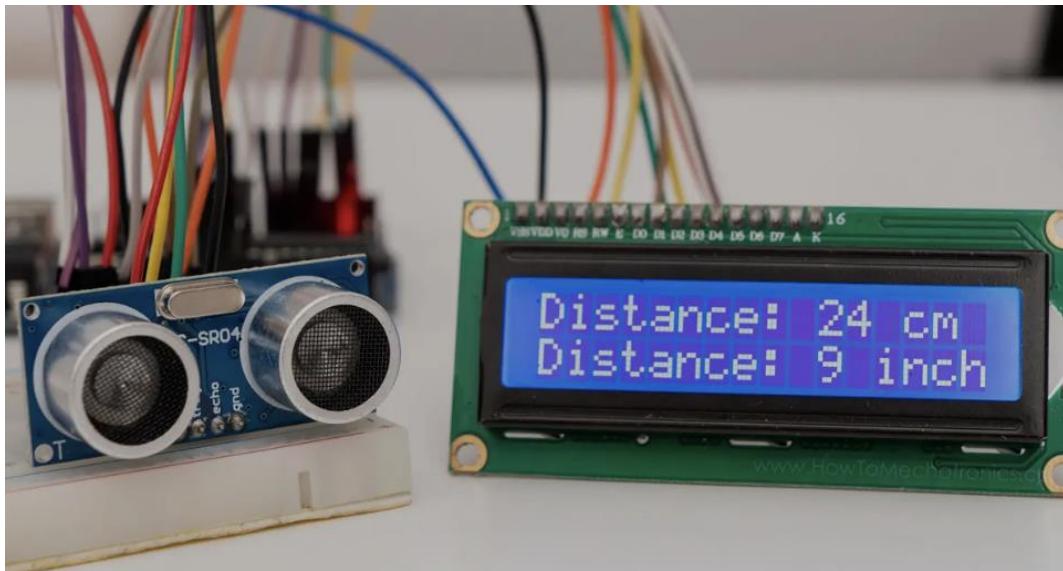
PASSIVE INFRARED SENSOR (PIR)



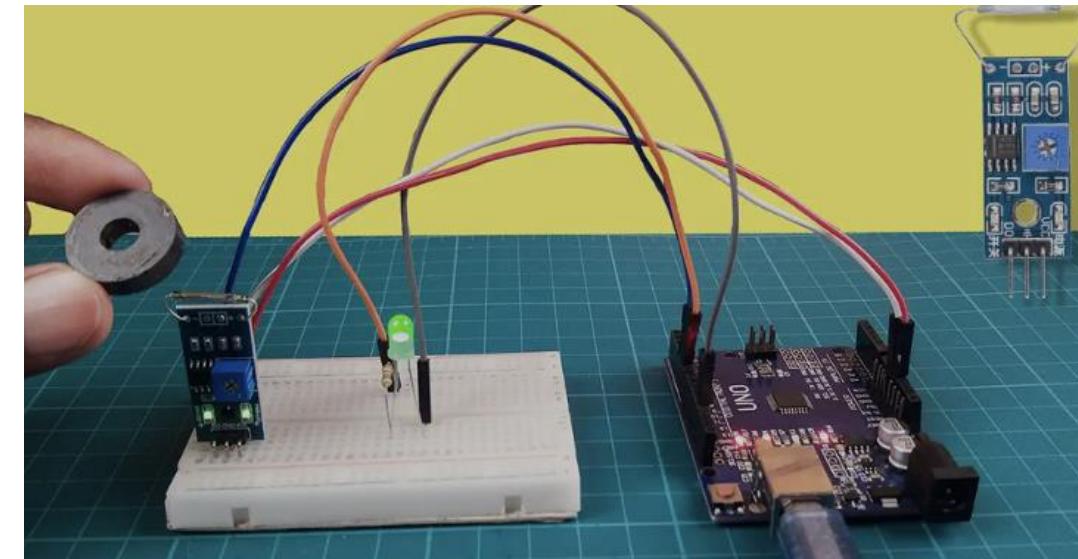
WATER LEVEL SENSOR



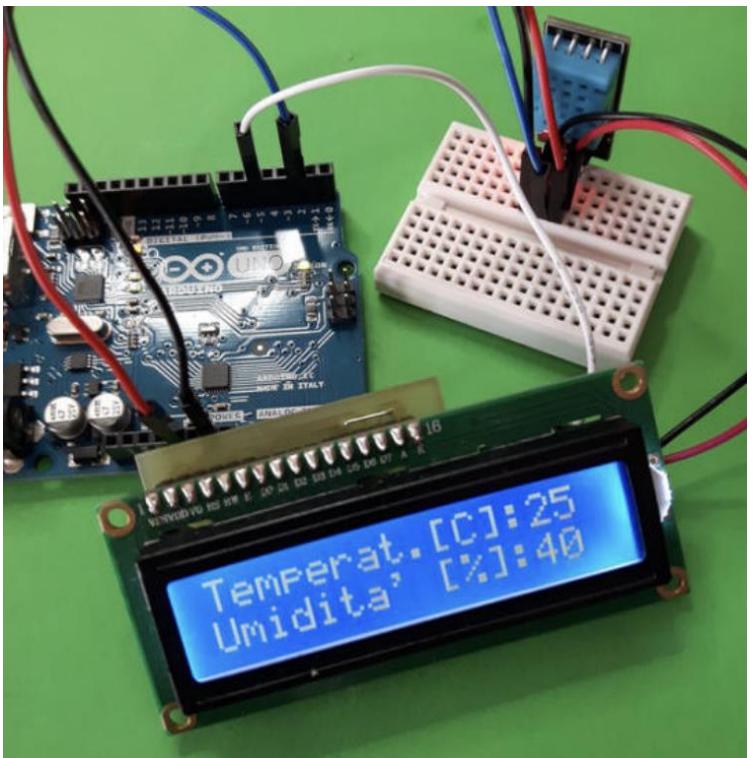
ULTRASONIC SENSOR



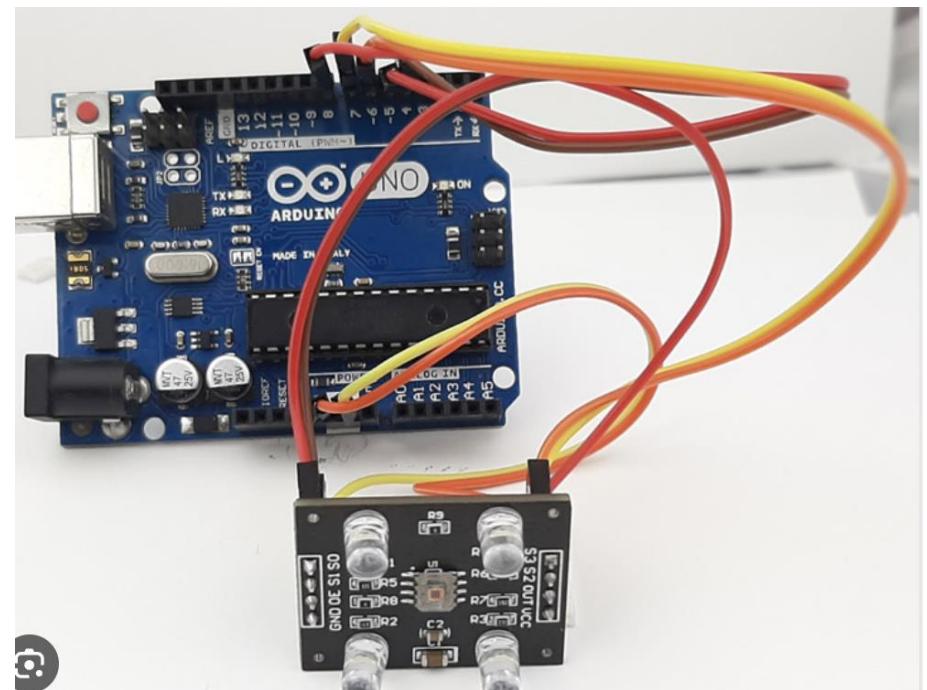
MAGNETIC REED SENSOR



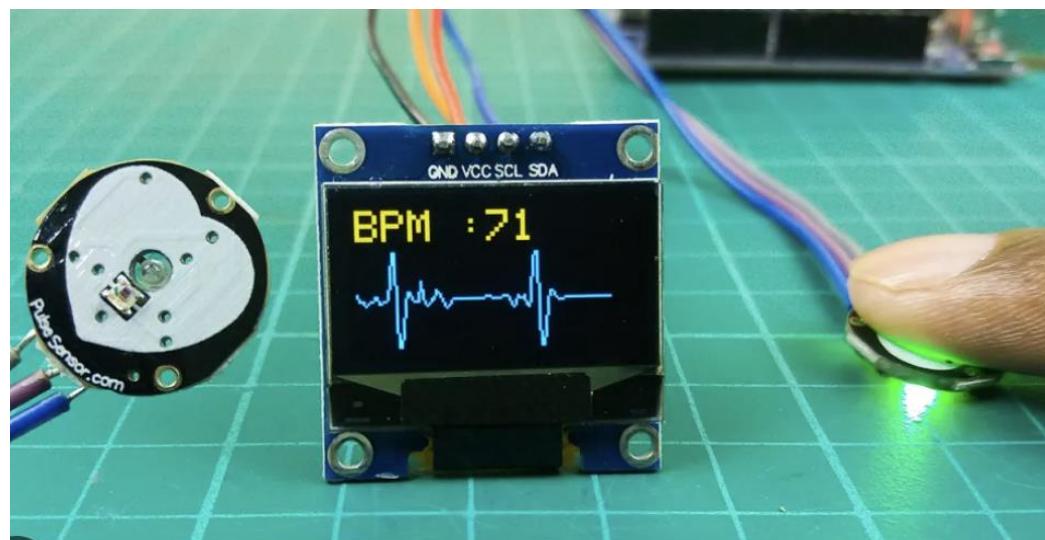
TEMPERATURE SENSOR



COLOR SENSOR



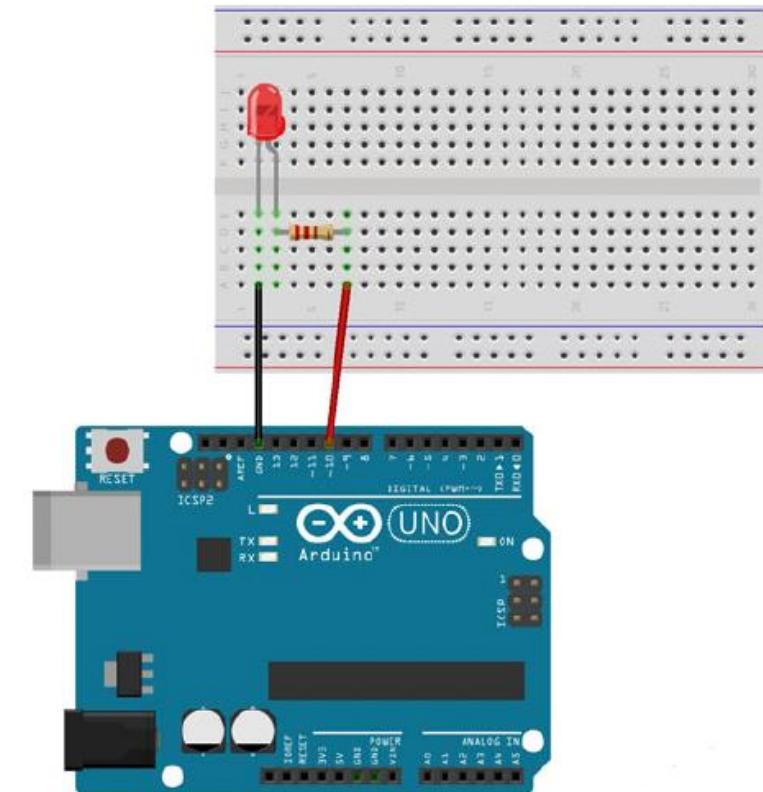
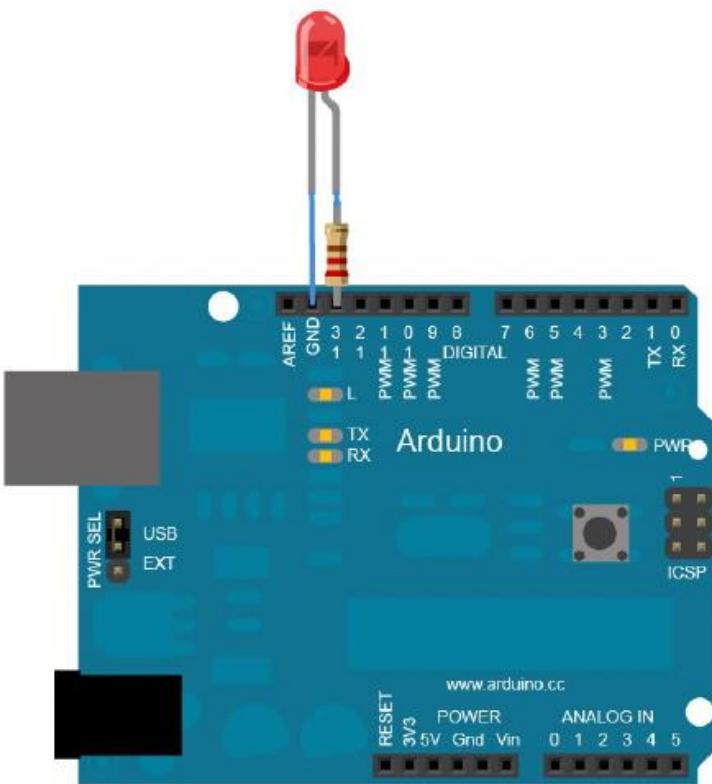
HEART BEAT SENSOR



Actuators (output)

- An **actuator** is a mechanism by which a control system acts upon an environment. The control system can be simple (a fixed mechanical or electronic system), software-based (e.g. a printer driver, robot control system), a human, or any other input
- It requires a control signal and a source of energy
- The control signal is relatively low energy and may be electric voltage or current, pneumatic or hydraulic pressure, or even human power.
- In the case of Arduino, the control signal can be provided through one or more digital outputs

The simplest actuator: a LED turned on and off



Sketch: blinking an external LED

```
int Time=1000;
int pinLed=4;

void setup() {
    pinMode(pinLed,OUTPUT); .on (LED)
}

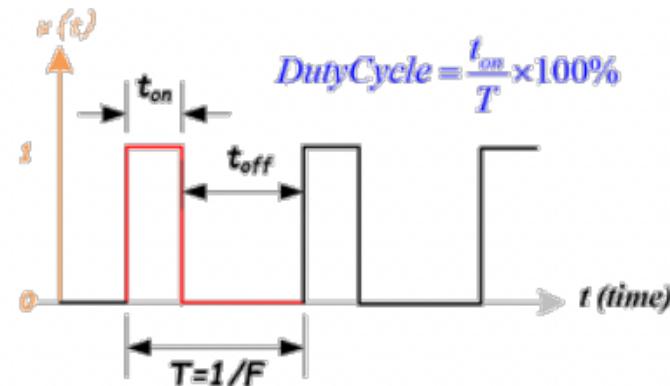
void loop() {
    digitalWrite(pinLed,HIGH); //LED on
    delay(Time);
    digitalWrite(pinLed,LOW); //LED off
    delay(Time);
}
```

Analog actuators

- In many situations we want to control an actuator using an analog signal (i.e. a continuous range of values)
- For instance, we may want to control LED's light intensity (dimming)
- Need to provide an adjustable voltage
- Can use Digital to Analog (D/A) converters, but a simpler alternative is...

Pulse Width Modulation (PWM)

- Generate a square wave, i.e. a signal switching between ON and OFF (still a digital signal!) with a given **duty cycle**:



- If the period is short enough, actuators only see the mean value:

$$V_{mean} = d \cdot V = \frac{t_{ON}}{T} \cdot V$$

- Can get different mean values by setting different duty cycles, d
- This is called **Pulse Width Modulation (PWM)**
- NB: T should be much shorter than the intended time scale of variation of the output signal

PWM control

- A square wave can be simulated via software, by alternating ON and OFF states
- However, in typical MCU boards there are output pins which can generate square waves via hardware, by exploiting the internal clock signal
- In Arduino UNO, pins 3, 5, 6, 9, 10, 11 can be configured for PWM output

PWM control through Arduino

- PWM control is done with:

```
analogWrite(myPWMpin, value)
```

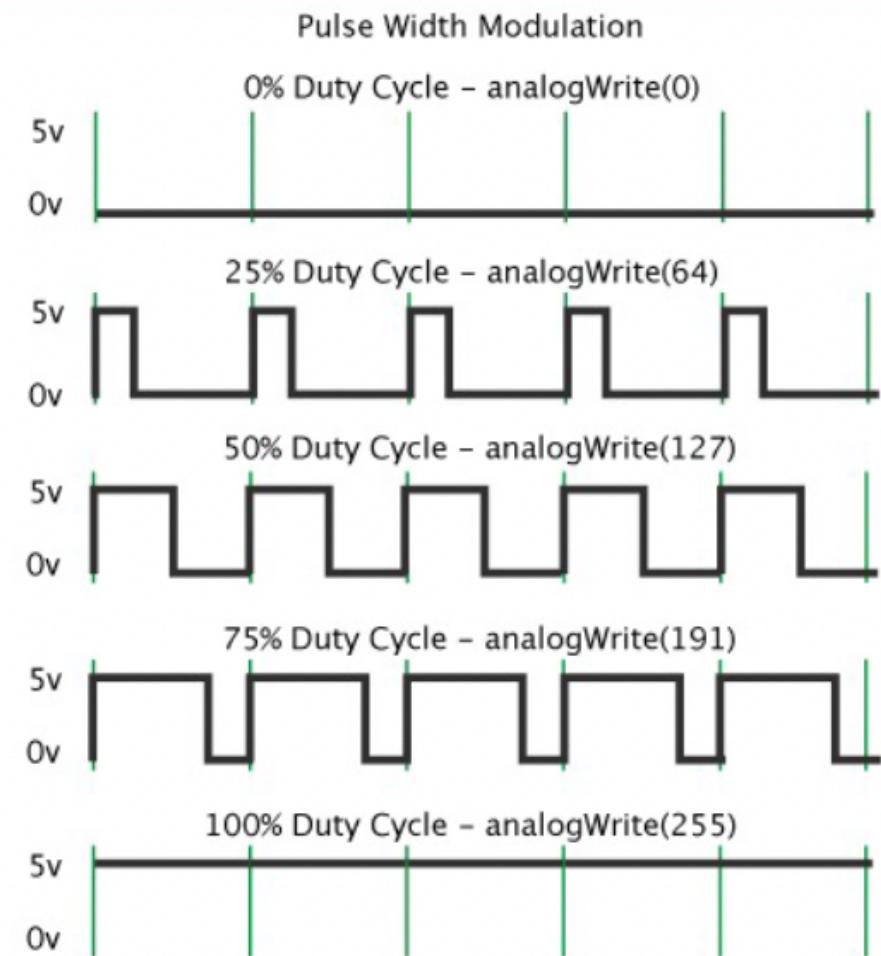
where **value** is an 8-bit value (hence 0-255) which is compared against an 8-bit counter.

When the counter is **less than** the PWM **value**, the pin output is **HIGH**

- In this way:

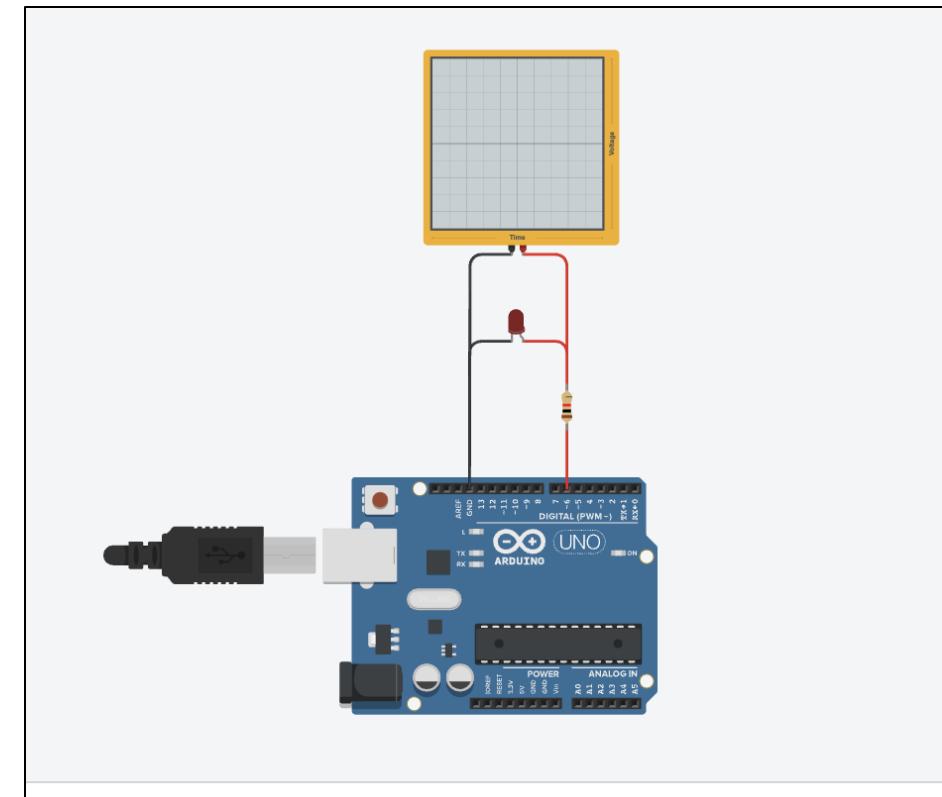
value=0 means 0% duty cycle

value=255 means 100% duty cycle



Sketch: dimming a LED

```
int Time = 10;  
int pinLed = 3; // PWM Pin (3,5,6,9,10,11)  
  
void setup() {  
    pinMode(pinLed,OUTPUT);  
}  
  
void loop() {  
    for(int k = 0;k<=255;k++){  
        analogWrite(pinLed,k);  
        delay(Time);  
    }  
  
    for(int k=255;k>=0>k--){  
        analogWrite(pinLed,k);  
        delay(Time);  
    }  
}
```

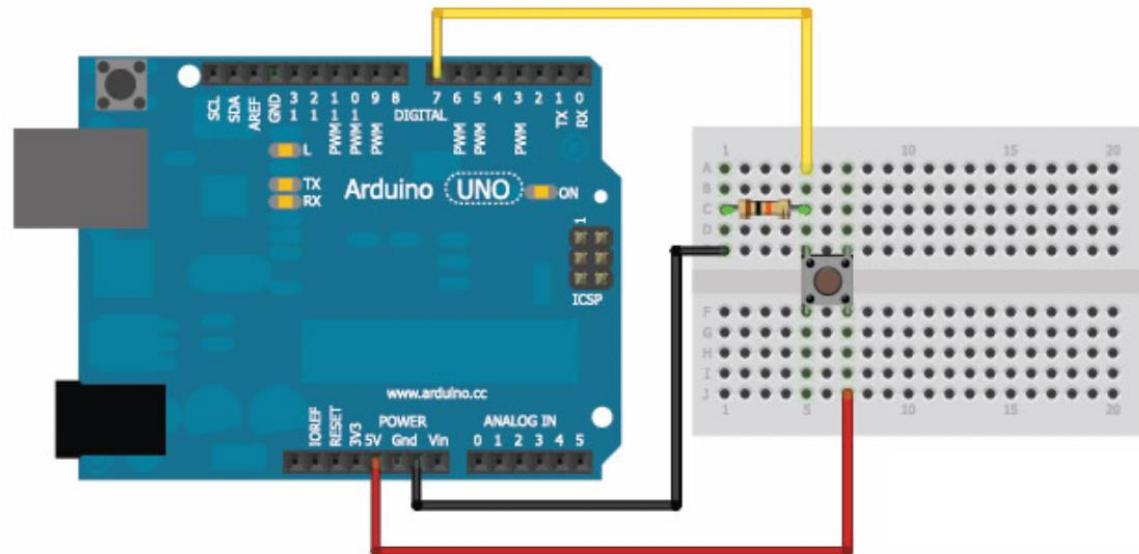


Sensors (input)

- A **sensor** is a converter that measures a physical quantity and converts it into a signal which can be read by an observer or by an (today mostly electronic) instrument
- Like actuators, sensors can be
 - Digital
 - Analog

Digital sensors

- **Digital sensors** just act as switches
- Can be either **on** or **off**
- Example: a button

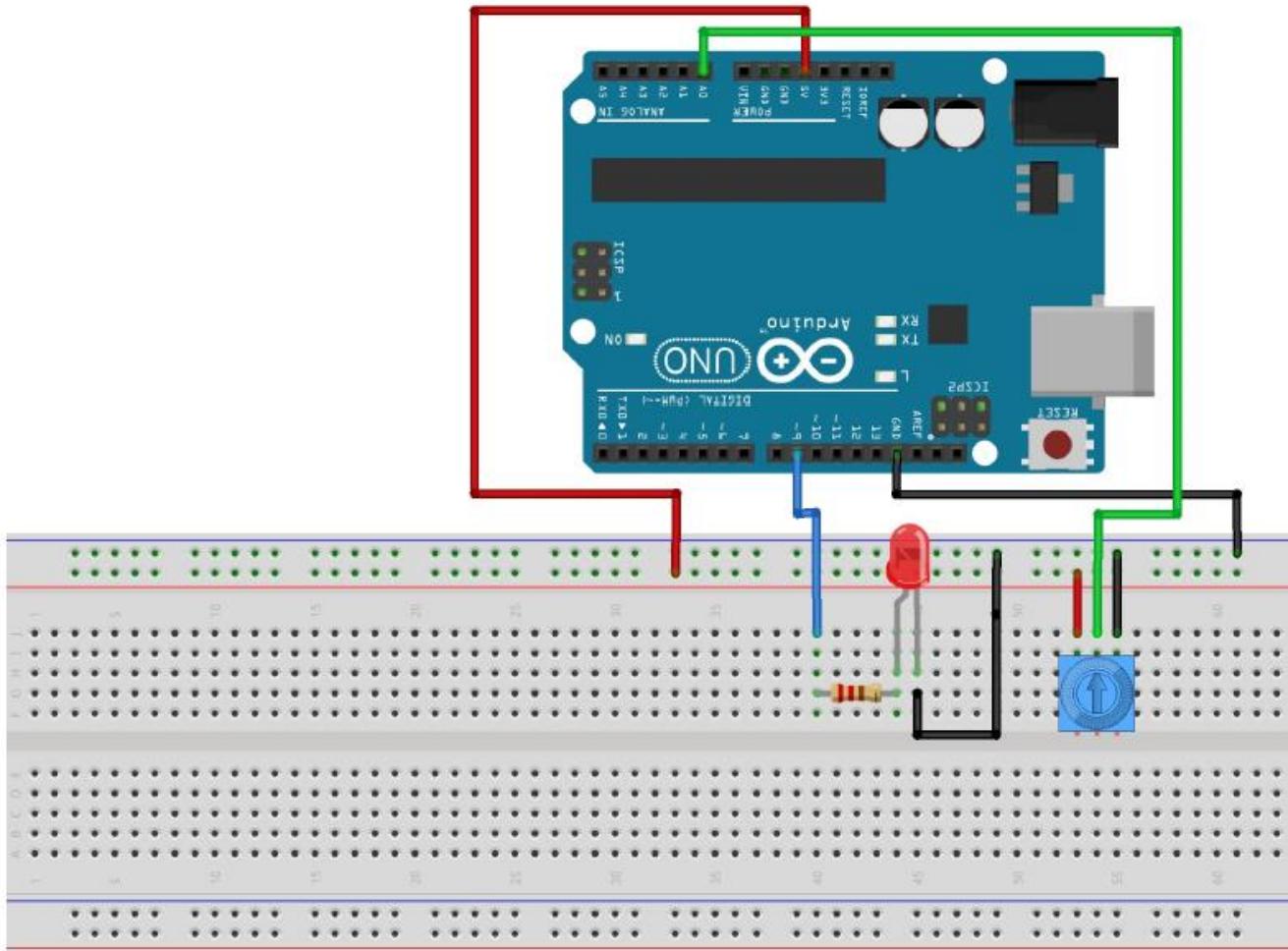


```
// constants won't change. They're used here to set pin numbers:  
const int buttonPin = 2; // the number of the pushbutton pin  
const int ledPin = 13; // the number of the LED pin  
// variables can change their values:  
int buttonState = 0; // variable for reading the pushbutton status  
  
void setup() {  
pinMode(ledPin, OUTPUT); // initialize the LED pin as an output  
pinMode(buttonPin, INPUT); // initialize pushbutton pin as input...  
}  
  
void loop() {  
buttonState = digitalRead(buttonPin); // read state of the pushbutton  
// check if the pushbutton is pressed...  
if (buttonState == HIGH){  
digitalWrite(ledPin, HIGH); // turn LED on  
}  
else {  
digitalWrite(ledPin, LOW); // turn LED off  
}  
}
```

Analog sensors

- **Analog sensors** vary widely in function
 - **Active**: generate a signal (a voltage) which reflects the measured quantity. May require amplification to match the voltage with the MCU analog input range
 - **Passive**: rely on passive electrical components (resistors, capacitors, inductances, etc) which change their properties (resistance, capacitance, inductance) in response to a stimulus (temperature, light, force, deformation, etc)... Need an external power source and some circuitry to translate these property changes into voltage changes
- In both cases, if we know how the **voltage** relates to the quantity under measurement, the measured voltage determines the magnitude of that quantity

Dimming a LED with potentiometer



Potentiometers provide a variable amount of resistance that changes as you manipulate it

Dimming a LED with potentiometer

Setup:

- Set potentiometer pin (e.g. A0) as INPUT
- Set pin LED as OUTPUT

Loop:

- Read potentiometer pin value (analogRead)
- Convert this value into an output command
- Set pin LED value (analogWrite)

***NB:** analogRead returns a 10-bit value (0-1023)
analogWrite wants a 8-bit value (0-255)

Dimming a LED with potentiometer

```
const int pin_led = 9;
const int pin_pot = 0;

int pot_value = 0;

void setup() {
    pinMode(pin_led,OUTPUT); // Pins configuration
}

void loop() {
    pot_value = analogRead(pin_pot); // Reading the potentiometer value
    int command = (pot_value * 256)/1024; // Elaboration of the signal to be used as control
    analogWrite(pin_led, command); // Control of the LED light
}
```

Controlling a servo motor with a potentiometer

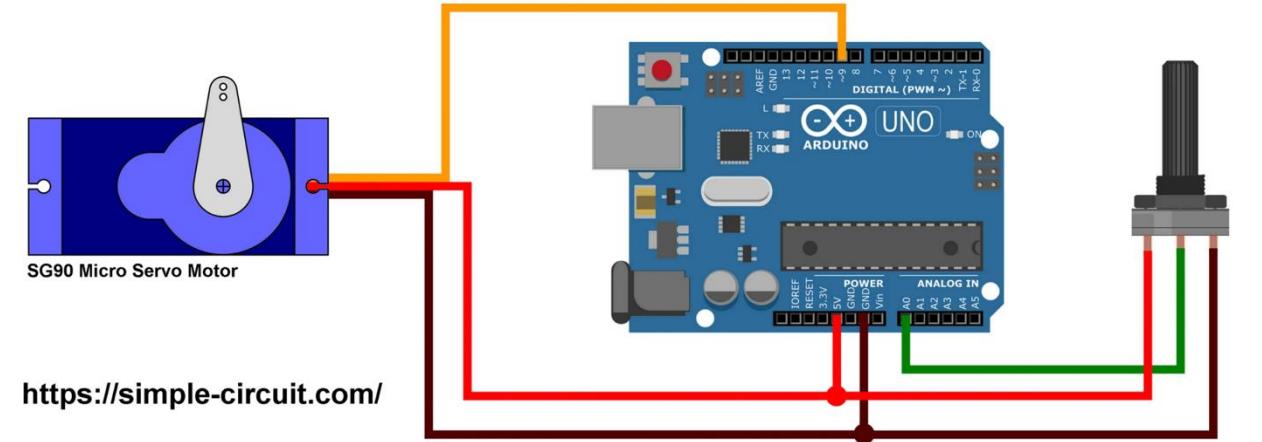
- Include servo library (Servo.h)

Setup:

- Set potentiometer pin (e.g. A0) as INPUT
- Set servo pin as OUTPUT
- Create Servo object

Loop:

- Read potentiometer pin value (analogRead)
- Convert this value into an output value
- Set potentiometer pin value (servo.write)



NB: `analogRead` returns a 10-bit value (0-1023)

Must convert it into an angle value (0° - 179°)

Controlling a servo motor with potentiometer

```
#include <Servo.h> //Servo library
Servo myservo; //Object servo

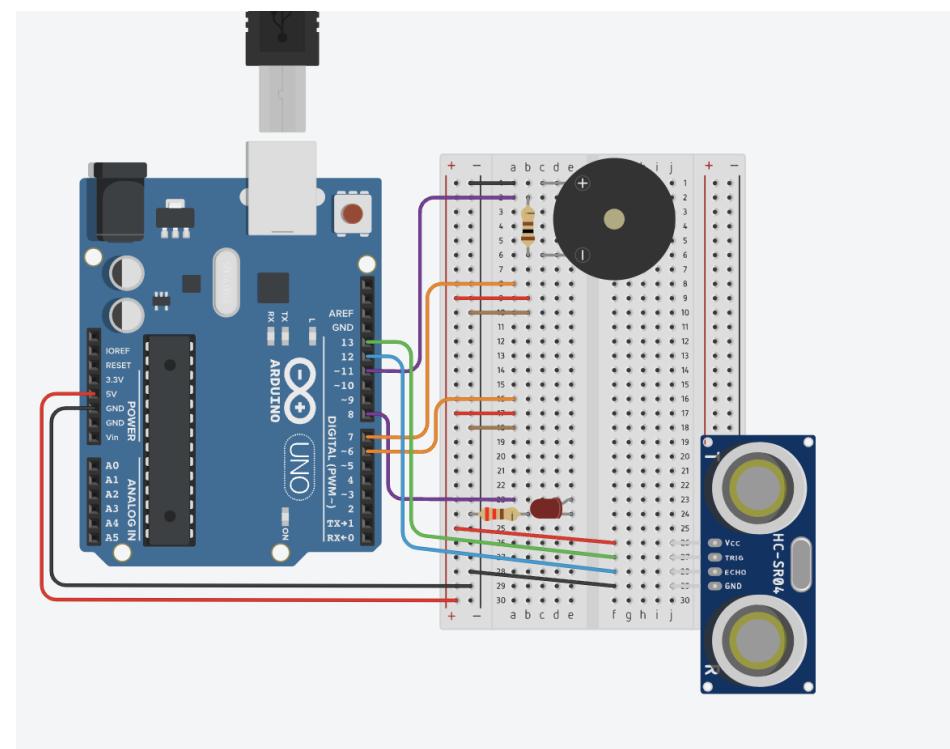
const int pin_pot = 0;
const int pin_servo = 9;

int pot_value = 0;

void setup() {
    myservo.attach(pin_servo);
    pinMode(pin_pot, INPUT);
}

void loop() {
    pot_value = analogRead(pin_pot); // Reading the potentiometer value
    int command = map(pot_value, 0, 1023, 0, 179); // Elaboration of the signal to be used as control
    myservo.write(command); // Control of the servo shaft position
}
```

Ultrasonic Range Finder

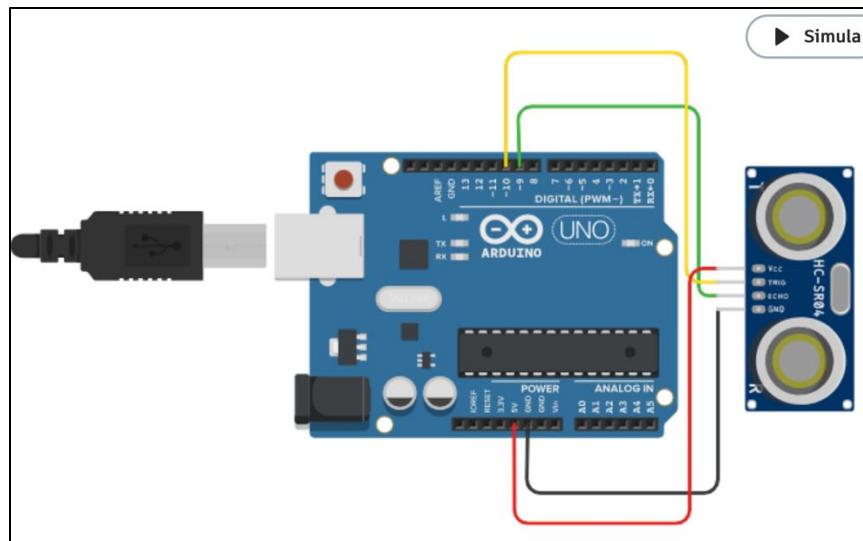


Ultrasonic Range Finder

The SEN136B5B is an ultrasonic range finder from Seeedstudio.

It detects the distance of the closest object in front of the sensor (from 3 cm up to 400 cm). It works by sending out a burst of ultrasound and listening for the echo when it bounces off of an object. It pings the obstacles with ultrasound.

The Arduino board sends a short pulse to trigger the detection, then listens for a pulse on the same pin using the `pulseIn()` function. The duration of this second pulse is equal to the time taken by the ultrasound to travel to the object and back to the sensor. Using the speed of sound, this time can be converted to distance.



```
const int pingPin = 7;

void setup() {
  // initialize serial communication:
  Serial.begin(9600);
}

void loop() { long duration, inches, cm;
  pinMode(pingPin, OUTPUT);
  digitalWrite(pingPin, LOW);
  delayMicroseconds(2);
  digitalWrite(pingPin, HIGH);
  delayMicroseconds(5);
  digitalWrite(pingPin, LOW);

  pinMode(pingPin, INPUT);
  duration = pulseIn(pingPin, HIGH);
  inches = microsecondsToInches(duration);
  cm = microsecondsToCentimeters(duration);

  Serial.print(inches);
  Serial.print("in, ");
  Serial.print(cm);
  Serial.print("cm");
  Serial.println();

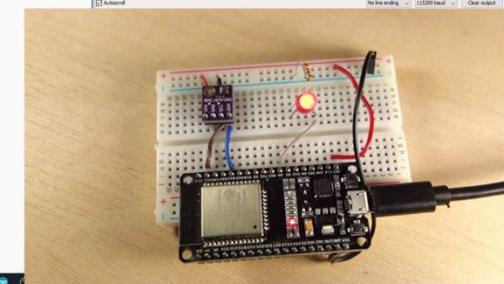
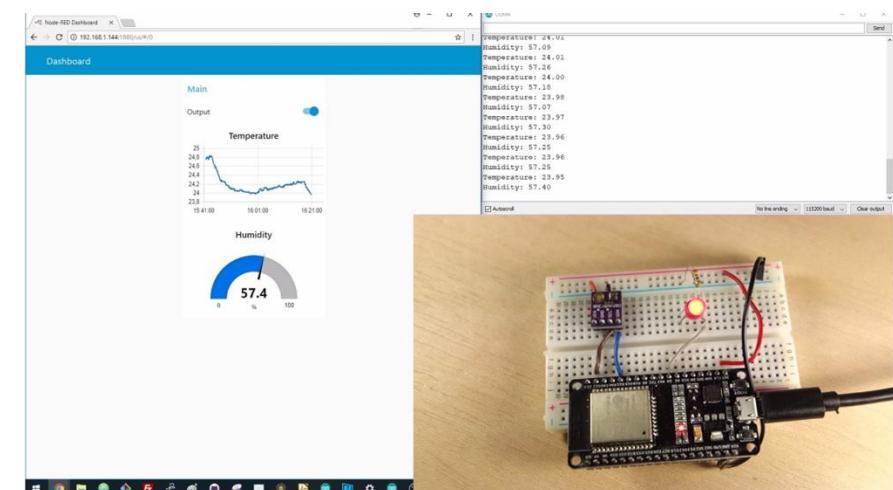
  delay(100);
}
```

```
long microsecondsToInches(long microseconds) {
  // According to Parallax's datasheet for the PING()), there are 73.746
  // microseconds per inch (i.e. sound travels at 1130 feet per second).
  // This gives the distance travelled by the ping, outbound and return,
  // so we divide by 2 to get the distance of the obstacle.

  return microseconds / 74 / 2;
}

long microsecondsToCentimeters(long microseconds) {
  // The speed of sound is 340 m/s or 29 microseconds per centimeter.
  // The ping travels out and back, so to find the distance of the object we
  // take half of the distance travelled.

  return microseconds / 29 / 2;
}
```



```
#include <ArduinoMqttClient.h>
#include <ArduinoJson.h>
#include <WiFi101.h>

char ssid[] = "..."; // SSID (name)
char pass[] = "..."; // network password
WiFiClient wifiClient;
MqttClient mqttClient(wifiClient);
const char broker[] = "test.mosquitto.org";
int port = 1883;
//set interval for sending messages (milliseconds)
const long interval = 10000;
unsigned long previousMillis = 0;
int count = 0;
```

```
void setup() {
  //Initialize serial and wait for port to open:
  Serial.begin(9600);
  while (!Serial) {
    ; // wait for serial port to connect.
  }
  // attempt to connect to Wifi network:
  Serial.print("Attempting to connect to WPA SSID: ");
  Serial.println(ssid);
  while (WiFi.begin(ssid, pass) != WL_CONNECTED) {
    // failed, retry
    Serial.print(".");
    delay(5000);
  }
  Serial.println("You're connected to the network");
  Serial.println();
  Serial.print("Attempting to connect to the MQTT broker: ");
  Serial.println(broker);
  if (!mqttClient.connect(broker, port)) {
    Serial.print("MQTT connection failed! Error code = ");
    Serial.println(mqttClient.connectError());
    while (1);
  }
  Serial.println("You're connected to the MQTT broker!");
  Serial.println();
}
```

```
void loop() {
    // call poll() regularly to allow the library to send MQTT keep alives which
    // avoids being disconnected by the broker
    mqttClient.poll();

    unsigned long currentMillis = millis();

    if (currentMillis - previousMillis >= interval) {
        // save the last time a message was sent
        previousMillis = currentMillis;

        //record random value from A0, A1 and A2
        //int Rvalue = analogRead(A0);
        //int Rvalue2 = analogRead(A1);
        //int Rvalue3 = analogRead(A2);

        Serial.print("Sending message to /dibris/indoor: ");
        StaticJsonDocument<256> doc;
        doc["name"] = "2";
        doc["x"] = "44.40352";
        doc["y"] = "8.9725";
        doc["t"] = random(22,40);
        char buffer[256];
        serializeJson(doc, buffer);

        mqttClient.beginMessage("/dibris/indoor");
        mqttClient.print(buffer);
        mqttClient.endMessage();

        Serial.println();
    }
}
```