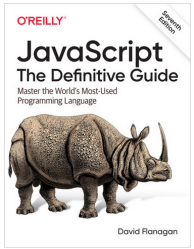
Internet of Things (IOT)

IoT Programming

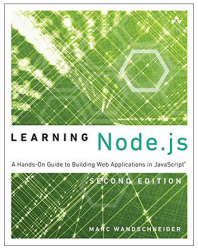
DIBRIS - Università di Genova

a.y. 2023-2024

Main references

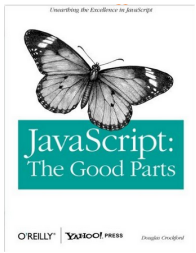


David Flanagan. JavaScript: The Definitive Guide, 7th Edition. O'Reilly Media, 2020

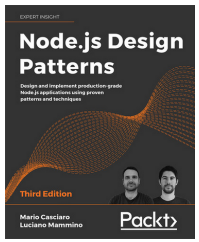


Marc Wandschneider. Learning Node.js, 2nd Edition. Addison-Wesley, 2016

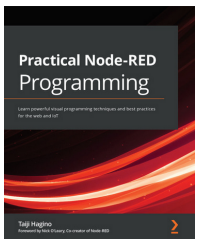
Other suggested books



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

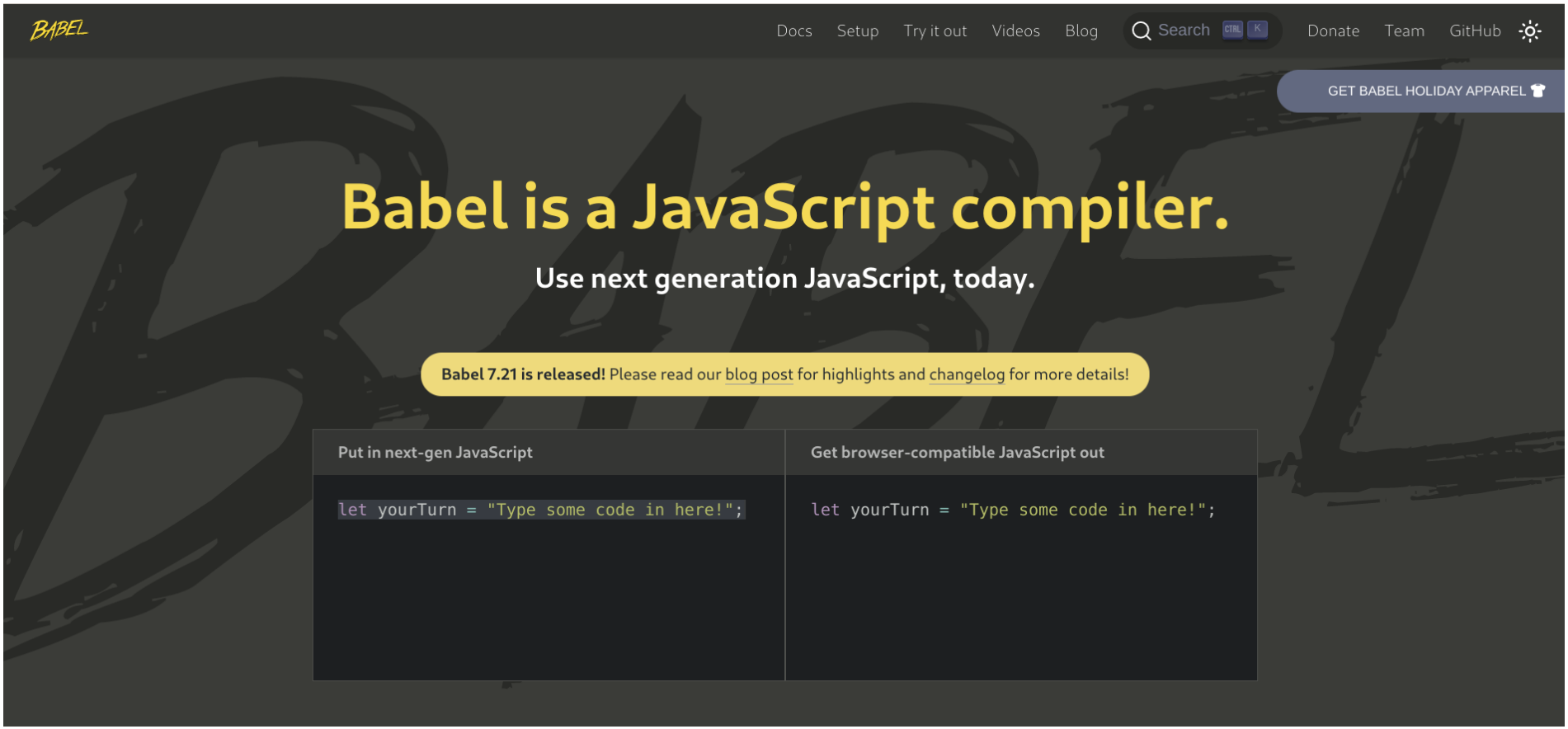
Historical overview

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 14.0 (June 2023)
- specs at `https://262.ecma-international.org/$V$/` with $\texttt{\$V\$} \in \{\texttt{5.1}, \texttt{6.0}, \ldots, \texttt{14.0}\}$

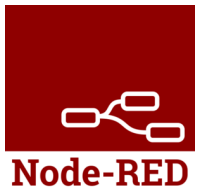
Lot of ES 5.1 legacy code!

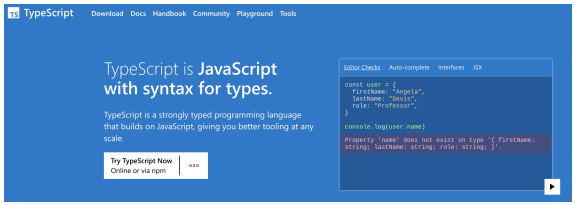
Useful projects to translate next-gen JS into ES 5.1



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

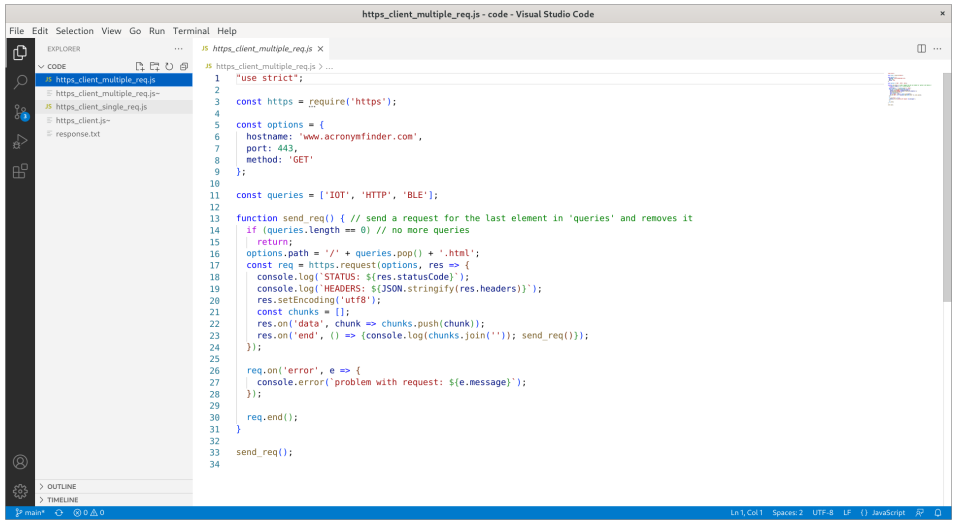
```
> console.log('Hello world!')
Hello world!
undefined
```

- scripts can be executed via the command line with `node`

```
$ node hello.js
```

Developing code in JavaScript

Visual Studio Code



- 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

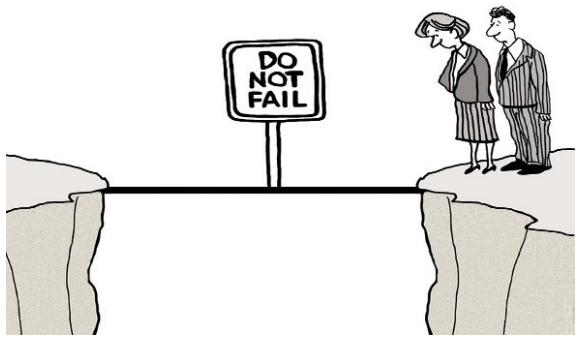
```
> return x;

Uncaught SyntaxError: Illegal return statement
```

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

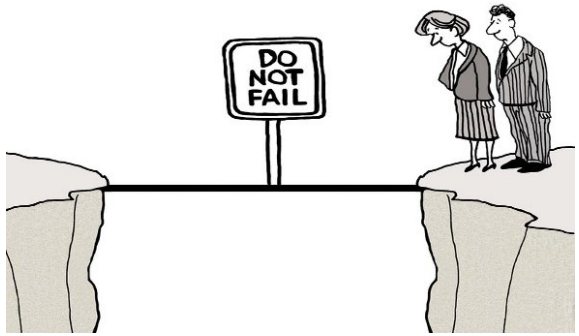
- JavaScript is very flexible, it rarely complains
- for safety reasons be sure to always run programs in strict mode

```
'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 from properties 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')
```

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
1023.9999999999999
```

Arithmetic operators

- Standard operators: `+`, `-`, `*`,`/`, `%`, `**`
- 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)
1.732050807568772
> Math.pow(3, 1/3)
1.4422495703074083
> Math.sin(0)
0
> Math.log(10)
2.302585092994046
> Math.exp(3)
20.085536923187668
```

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'
```

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

```
> "a string";
'a string'
> 'another string';
'another string'
> 'a double quote "';
'a double quote "'
> "a single quote '";
'a single quote \''
> console.log("\\ \' \n");
\ '

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 '
    [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 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['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 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)`

Example

```
> 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"}'
```

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 hardly needed
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.index=5;
5
> JSON.stringify(a);
'[1,2,3]'
```

Remark: in arrays only indexed values are stringified

Some useful array functions

push($e_0, \ldots, e_n$)/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($e_0, \ldots, e_n$)/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*, $item_1, \ldots, item_n$)

```
> 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

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 cat_all(...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 cat_all1(){
    let res='';
    for(let i=0;i<arguments.length;i++)res+=arguments[i]+' '
    return res;
};
undefined
> function cat_all2(){
   let res='';
   for(let i in arguments)res+=arguments[i]+' '
   return res;
}
undefined
> function cat_all3(){ // recall: arguments is not an array!
   return Array.from(arguments).join(' '); // only since ECMAScript 6
};
undefined
> cat_all1('a','b','c');
'a b c '
> cat_all2('a','b','c');
'a b c '
> cat_all3('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 then **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 then **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);
undefined
> cell.value;
42
> cell['set'](0);
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 (since ES6)

```
> let cell1={value:0,set(v){this.value=v}};
undefined
> let name='set'; // computed property name
undefined
> let cell2={value:0,[name+'_cell'](v){this.value=v}};
undefined
> cell2;
{ value: 0, set_cell: [Function: set_cell] }
> 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)

Symbols are opaque values

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

Symbols has property name (since ES6)

A safe extension mechanism (see iterators later on)

```
> function add_property(obj){
      let s=Symbol();
      obj[s]=42;
      return s;
  };
undefined
> let o={x:1,y:2}, new_prop=add_property(o);
undefined
> o[new_prop];
42
> Object.getOwnPropertySymbols(o);
[ Symbol() ]
>  Object.getOwnPropertySymbols(o)[0]===new_prop;
true
> Object.getOwnPropertyNames(o);
[ 'x', 'y' ]
> o.new_prop;
undefined
```

The `bind()` method

Example

```
> let obj={get_self(){return this;}};
undefined
> obj.get_self() === obj; // method invocation
true
> let f = obj.get_self;
undefined
> f===obj.get_self;
true
> f() === obj; // function invocation
false
> f(); // f isunbound from 'this'
undefined
```

Remark

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

The `bind()` method

It is possible to bind **`this`** in a function

```
> let get_obj=f.bind(obj); // binds 'this' in 'f' to 'obj'
undefined
> get_obj() === 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};
undefined
> let set_cell2=set.bind(cell2);
undefined
> set_cell2(42);
undefined
> cell2;
{ value: 42 }
> cell;
{ value: 0, set: [Function: set] }
```

The `call()` and `apply()` methods

Behavior

- f`.call(`o`,`a_1`,...,`a_n`)`: f is called on o with arguments a_1,...,a_n
- f`.apply(`o`,[`a_1`,...,`a_n`])`: arguments are passed through an array or 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);
undefined
> cell2;
{ value: 42 }
> set.apply(cell2,[43]);
undefined
> cell2;
{ value: 43 }
> cell2.set;
undefined
```

The `call()` and `apply()` methods

Example of "monkey-patching" (meta-programming)

```
> function log(o, m) { // o an object, m the key of a method of o
      let original = o[m];
      o[m] = function (...args) {
          console.log("Entering:", m);
          // `this` is the object on which is 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
Exiting: getAbs
42
undefined
```

Constructors

Functions can also work as constructors

```
// convention: constructors' names start with a capital letter
> function Cell(v){this.value=v};
undefined
> let cell=new Cell(42);
undefined
> cell;
Cell { value: 42 };
```

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) {
        if (this)
            this.value = n;
        return n;
    }
undefined
> new Num(42); // called as constructor
Num { value: 42 }
> Num(42); // called as a conversion function
42
```

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
```

Data and accessor properties

Example

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

- `v`: *data* property
- `val`: *accessor* property

Data and accessor properties

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)
- in this course we will mainly focus on data properties

Property descriptors

There are several ways to get 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(let i in a)console.log(i)
0
1
undefined
// returns an array of all own string-keys of 'a'
> Object.getOwnPropertyNames(a);
[ '0', '1', 'length' ]
// returns an array of all own enumerable string-keys of 'a'
> Object.keys(a);
[ '0', '1' ]
> let o={x:1,[Symbol()]:2};
undefined
// returns an array of all own symbol-keys of 'o'
> Object.getOwnPropertySymbols(o)
[ Symbol() ]
```

Property descriptors

Properties have attributes specified through descriptor objects

Example with data properties

```
> let a=[1,2];
undefined
// returns all own property descriptors of  'a'
> Object.getOwnPropertyDescriptors(a);
{
  '0': { value: 1, writable: true, enumerable: true, configurable: true },
  '1': { value: 2, 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 val(){return 'val:'+this.v;},
      set val(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 },
  val: {
    get: [Function: get val],
    set: [Function: set val],
    enumerable: true,
    configurable: true
  }
}
```

Attributes of data properties

Four attributes

- value: the value associated with the data property
- writable: is it possible to change its value?
- enumerable: is it visible to `for in` statement or `Object.keys()`?
- configurable: can we delete it, or arbitrarily change it 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 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`
- typical examples: arrays, strings, Set and Map objects, ...

Example

```
> function sumAll(iter) { // iter should be an iterable object
        let sum = 0;
        for (let i of iter) sum += i;
        return sum;
    };
undefined
> sumAll([1, 2, 3]);
6
> sumAll(new Set().add(1).add(2).add(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 an iterator object
- the key of the iterator method is always the symbol defined by `Symbol.iterator`
- 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]();
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(let 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:

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

Destructuring assignment

Examples

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

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

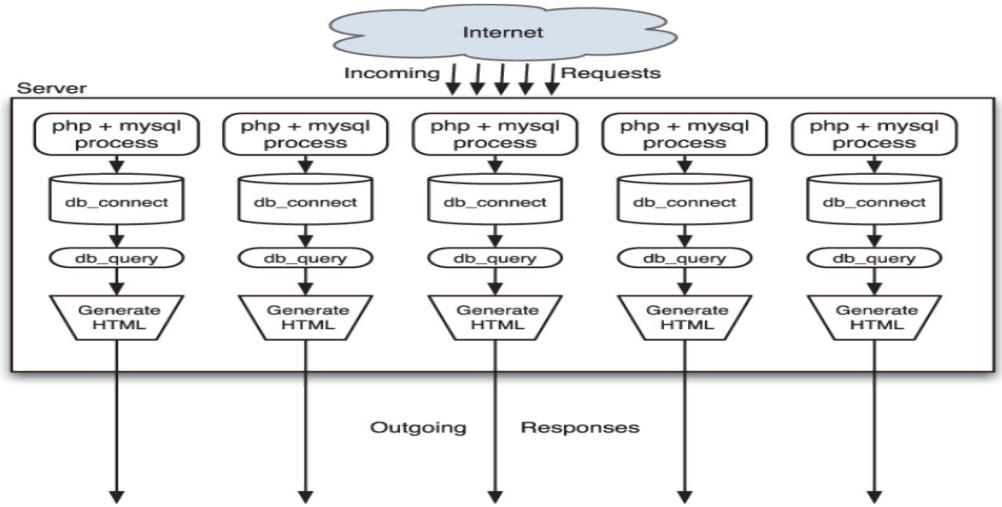
What is Node.js?

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` storing **`null`**, and `fd` the corresponding file descriptor, if the operation has succeeded
  - with `err` storing an `Error` object, if the operation has failed
- callback for read:

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

  After the `read` operation completes the callback is called by the system
  - with `err` storing **`null`**, `bytesRead` the number of read bytes, and the used buffer `buffer`, if the operation has succeeded
  - with `err` storing an `Error` object, if the operation has failed

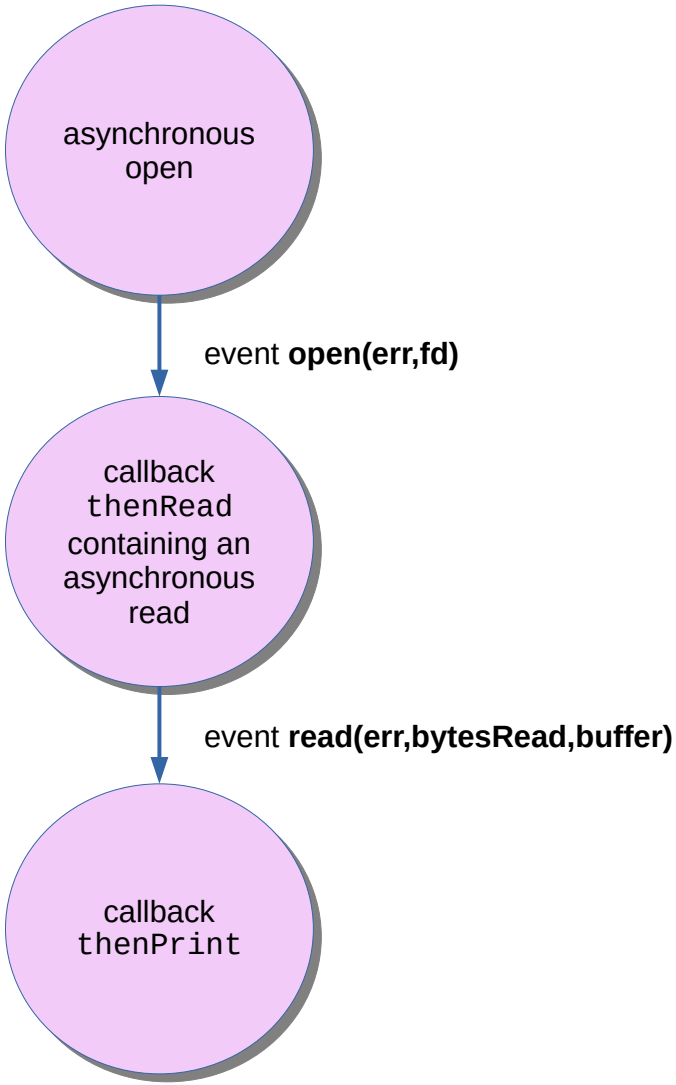
The Node.js style

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]}`);
```

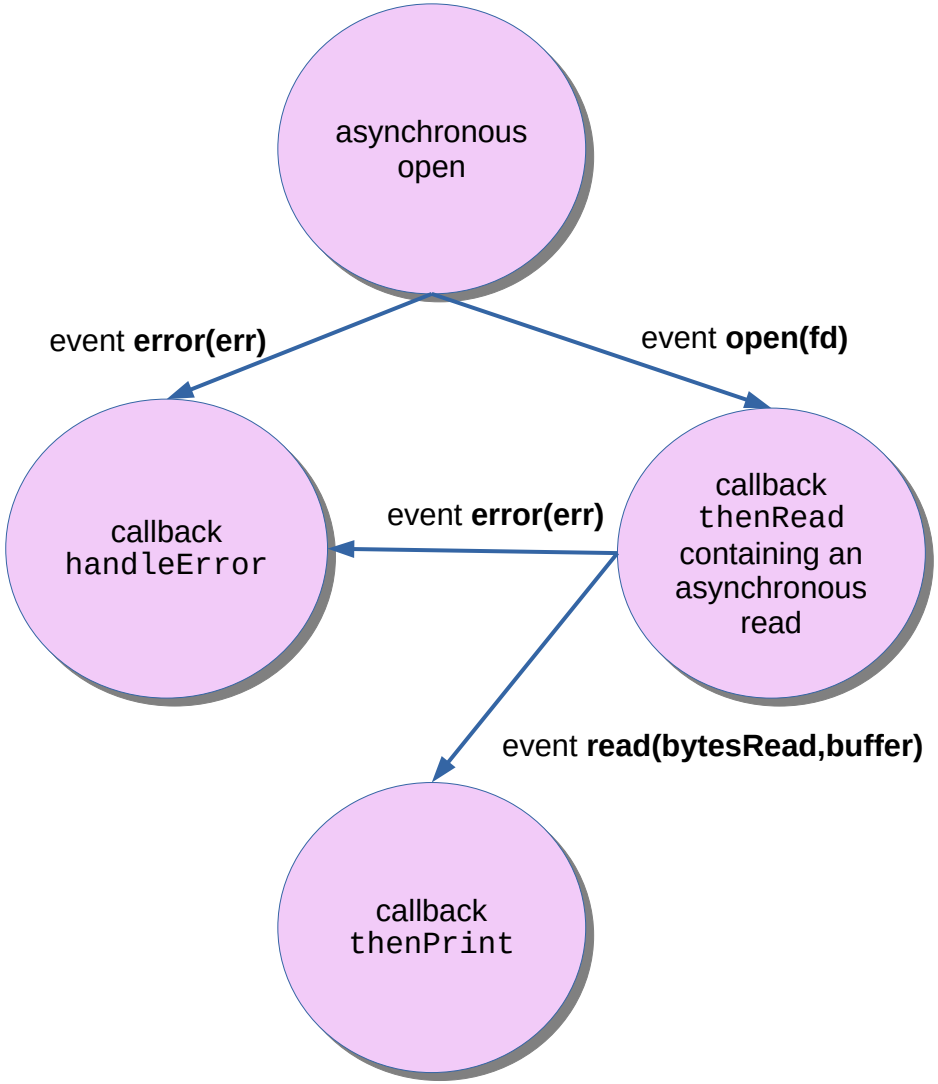
The Node.js style

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 Nodes.js

- nested functions
- closures

Example to recall scope rules

```
> function outer_level(){
    let level=1;
    function nested_level(){
       let level=2;
       return level;
    }
    return nested_level();
}
undefined
> outer_level();
2
```

`this`, `arguments` and nested functions

Example

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

        function 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 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 outer_level(){
    let level=1;
    function inc_level(){level++;}
    function nested_level(){
       let level=42;
       inc_level();
       return level;
    }
    return nested_level();
}
undefined
> outer_level();
```

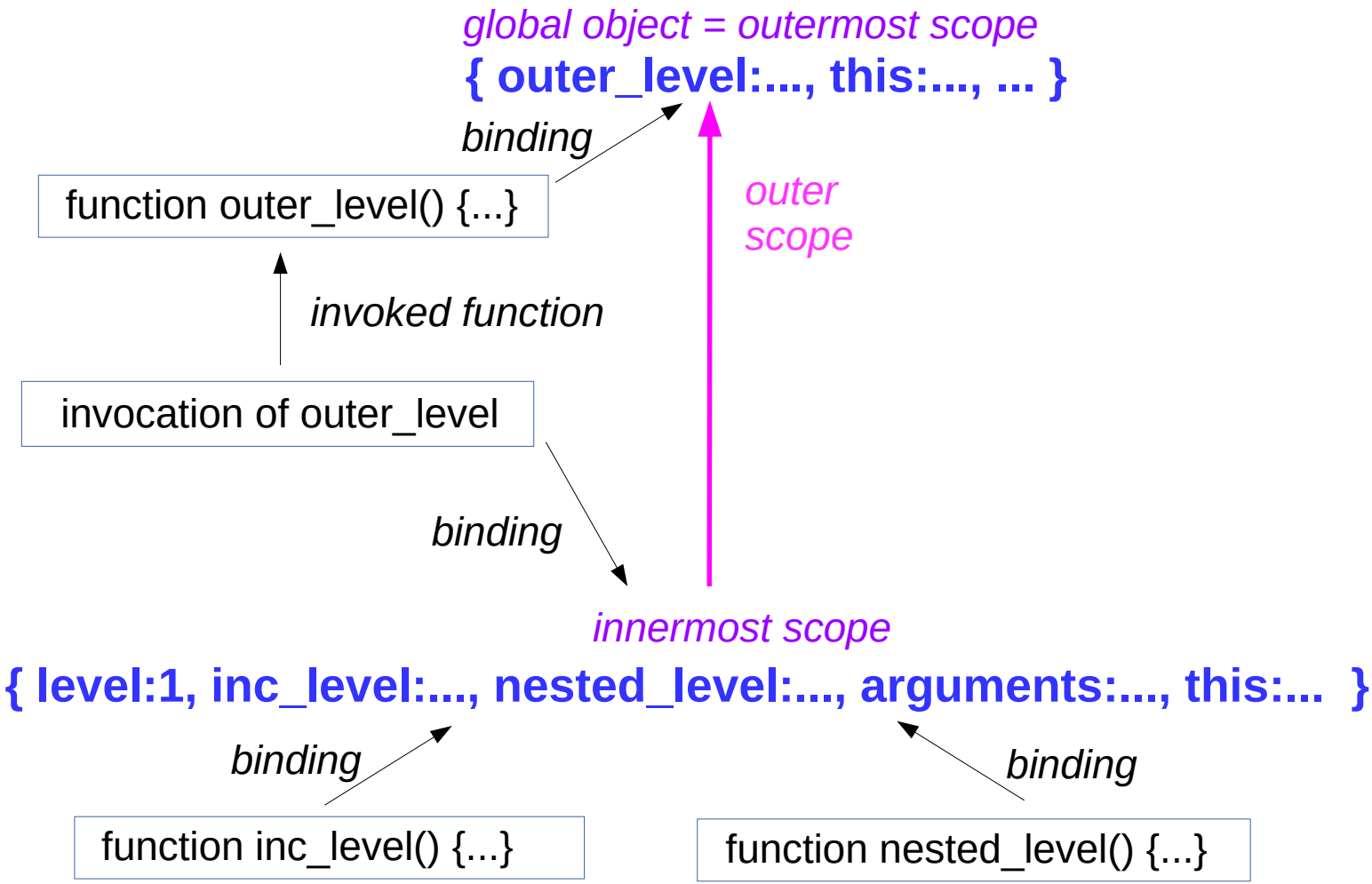
Nested functions

Example to recall scope rules

```
> function outer_level(){
    let level=1;
    function inc_level(){level++;};
    function nested_level(){
       let level=42;
       inc_level();
       return level;
    };
    return nested_level();
}
undefined
> outer_level();
42
```

The scope chain

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



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

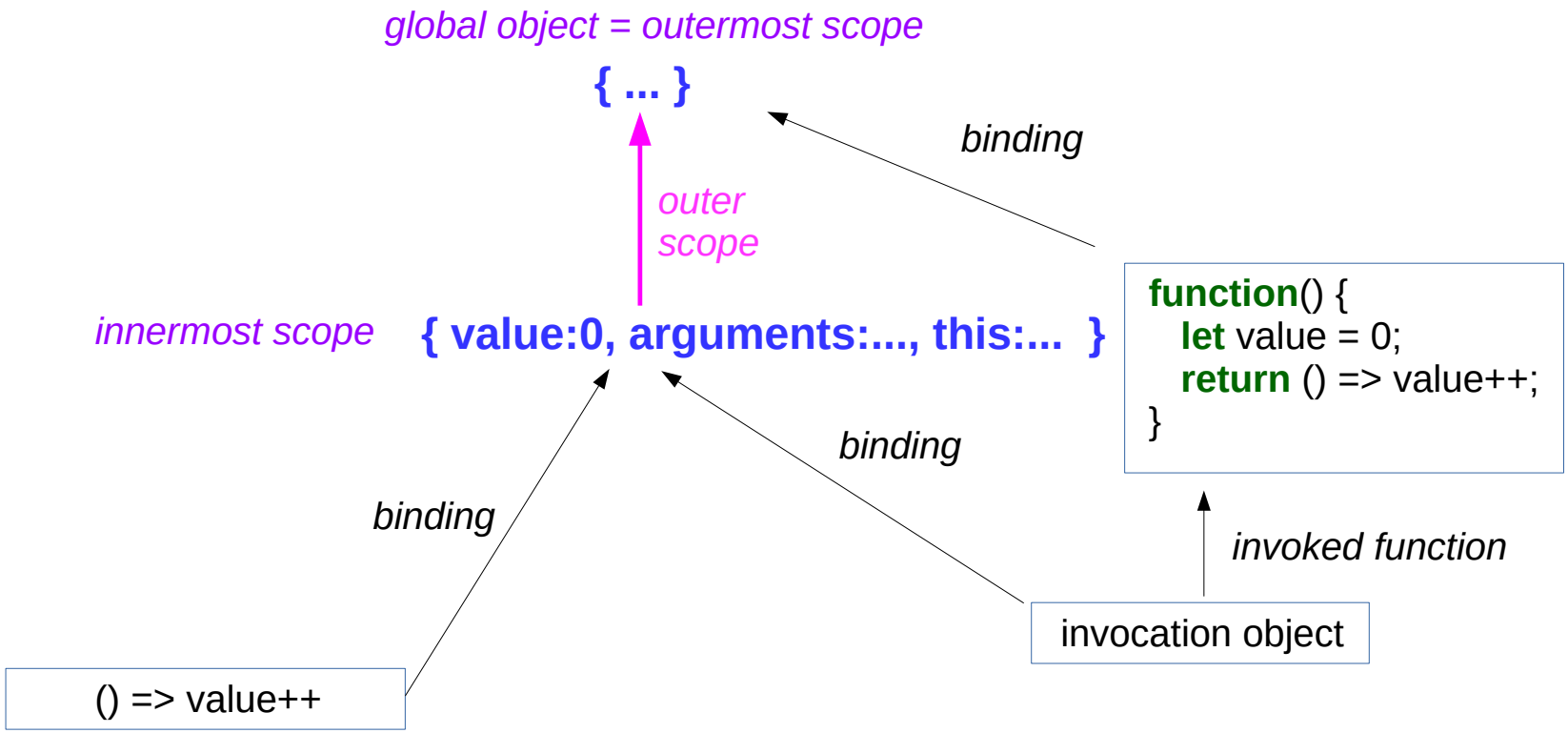
```
> let newInt = function () {
    let value = 0;
    return () => value++;
}();  // remark: the outer anonymous function is called with no args
undefined
> newInt
[Function]
> newInt()
0
> newInt()
1
```

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`

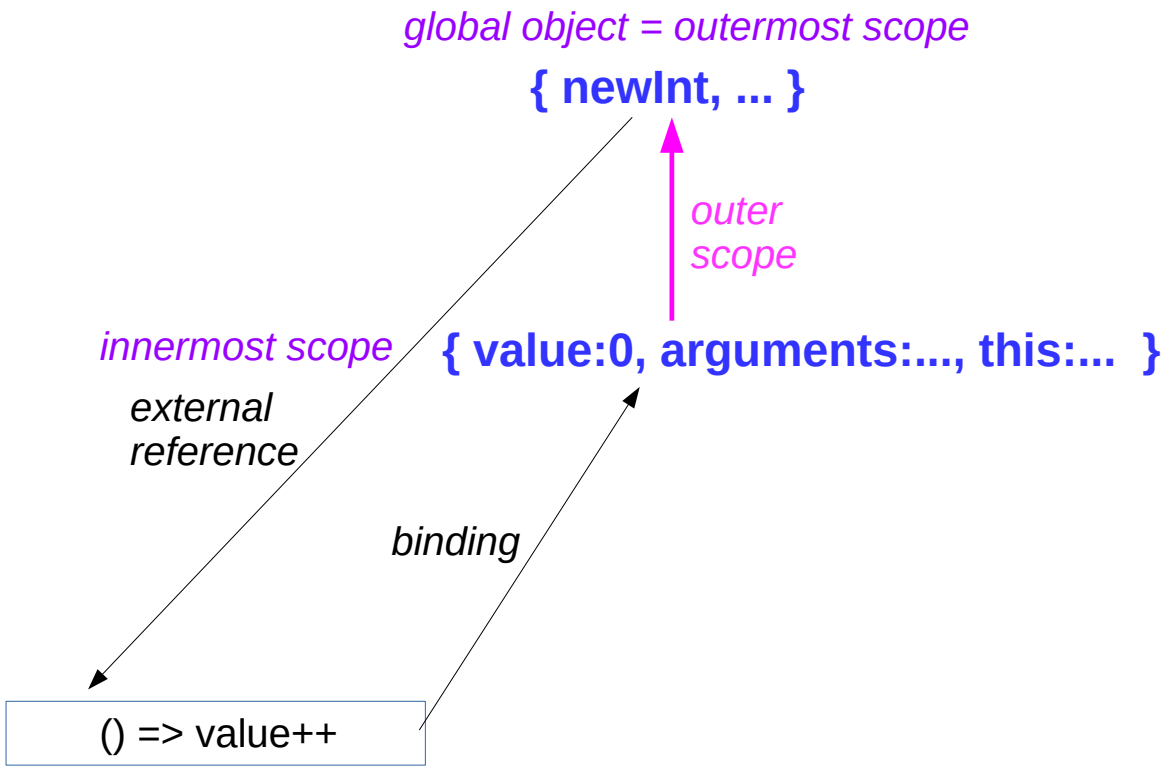
Pictorial view

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

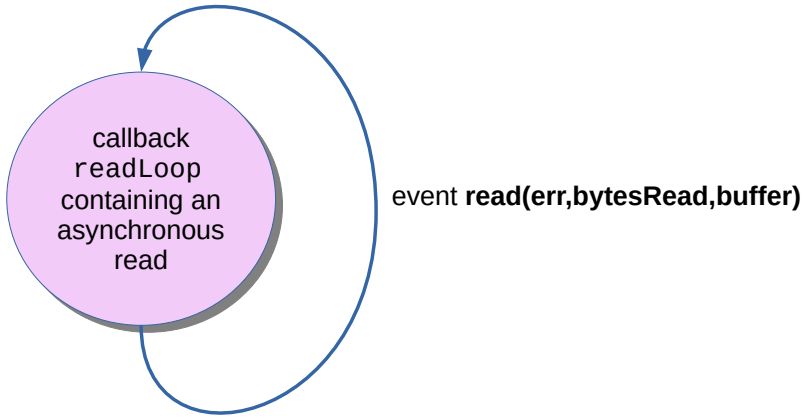
Non-standard recursion in Node.js patterns

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 callback executions depends on how async operations are scheduled

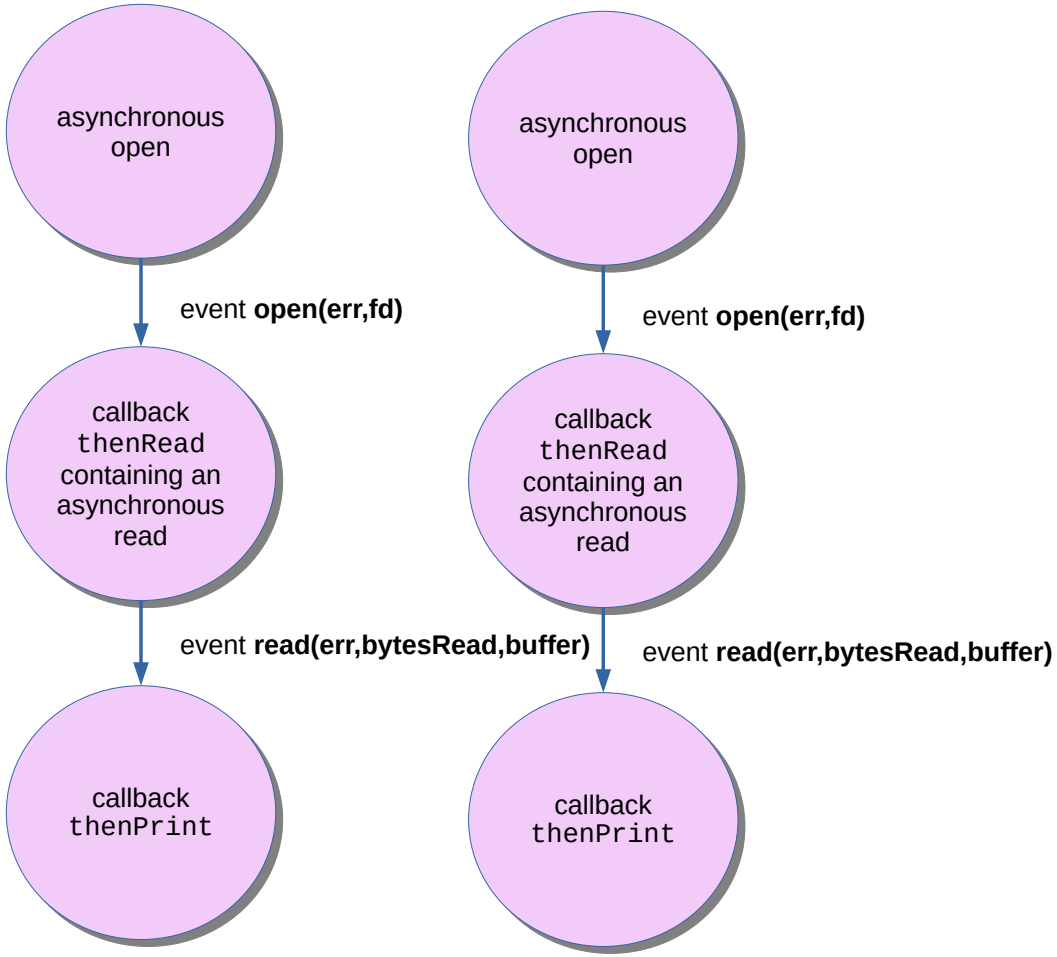
Which file will be printed first?

```
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 callback executions depends on how async operations are scheduled

Which file will be printed first?



The Node.js Event Loop

The event loop and the event/callback queues

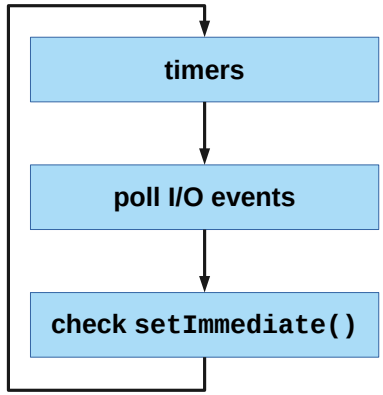
- Node.js runs in a single thread in a single process
- it executes a single event loop at time
- calls to asynchronous functions trigger events associated with callbacks
- pending events are enqueued with their corresponding callbacks
- Node.js exits only if the currently running code has finished executing and there are no pending events/callbacks
- Remark: if the currently running code does not terminate, then there will be no next loop to process the pending events!

The Node.js Event Loop

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
}
```

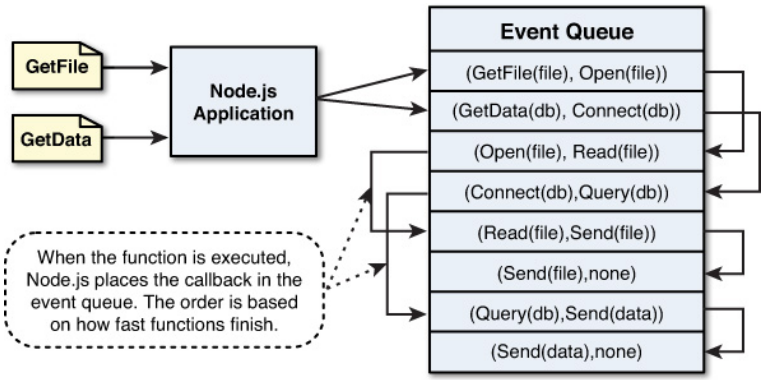
The Node.js Event Loop



A simplified overview

- there are several phases, with their own callback queue:
  - timers: handles callbacks scheduled with `setTimeout()`, `setInterval()`
  - poll: handles I/O events with their associated callbacks, controls timers
  - check: handles callbacks scheduled with `setImmediate()`
- callbacks in the queue of a phase are executed synchronously
- when the queue of a phase is empty or a limit is reached, the event loop moves to the next phase

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:
  1. if there are ready timers callbacks, then the event loop wraps back to the timers phase
  2. if there are callbacks scheduled by `setImmediate()`, then the event loop skips to the check phase
  3. 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
- use promises (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));
  res.on('end', () => console.log(chunks.join('')));
});

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

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

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));
  res.on('end', () => console.log(chunks.join('')));
});

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

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

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(`${JSON.stringify({ error: err.message })}\n`);
                }
                else {
                    res.writeHead(200, headers);
                    res.end(`${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(`${JSON.stringify(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

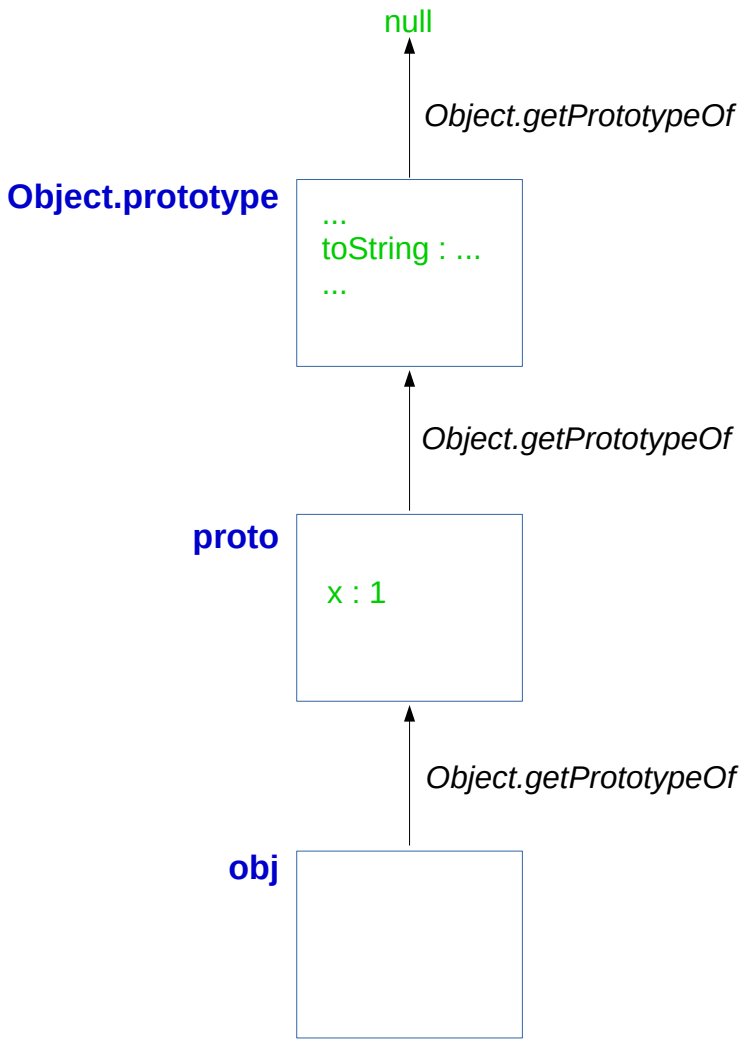
Prototype-based inheritance: the prototype chain

- 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

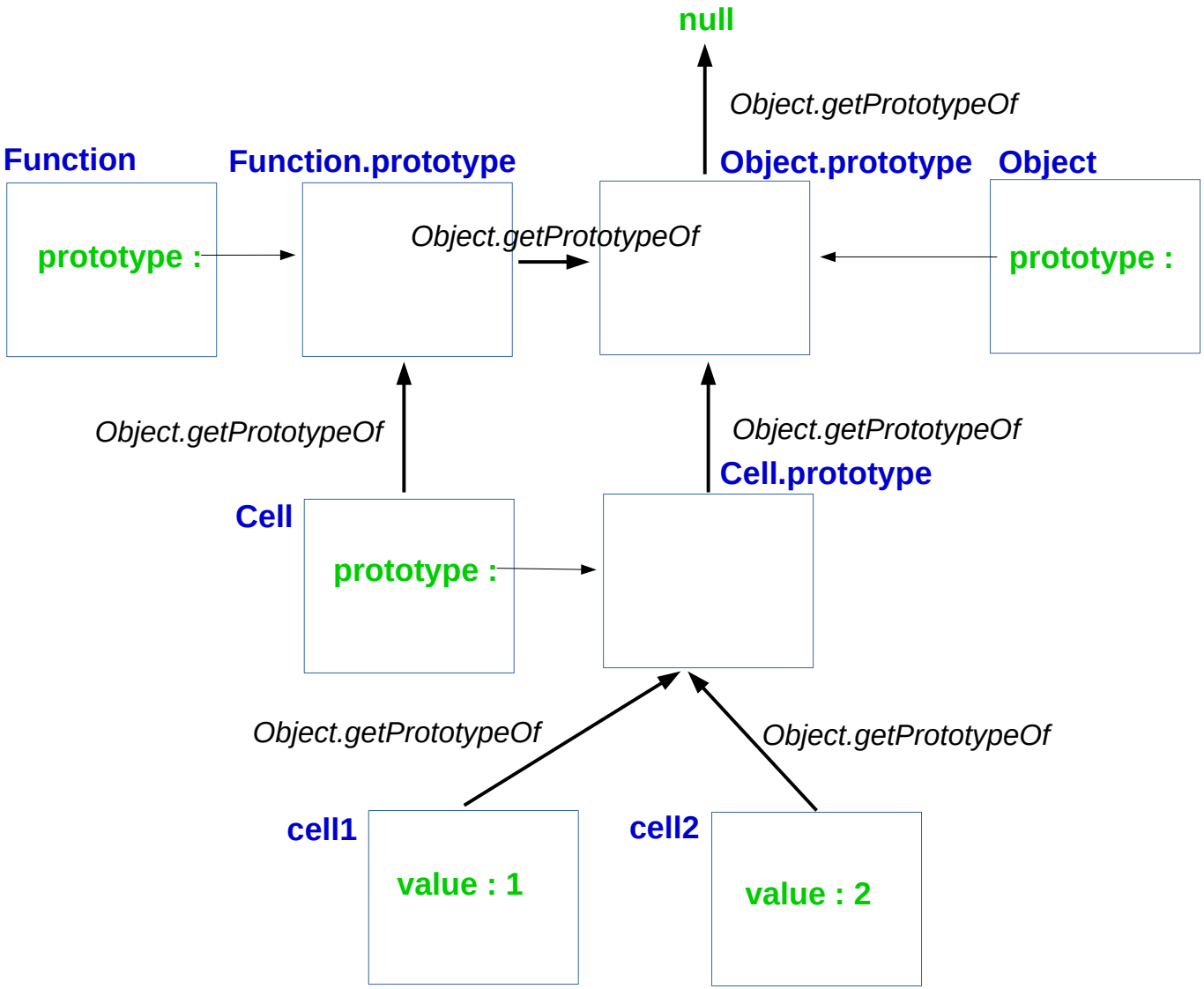
Every non-arrow function has the predefined `prototype` property

```
> 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
```

Remark

`prototype` is non-writable, non-enumerable and non-configurable

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

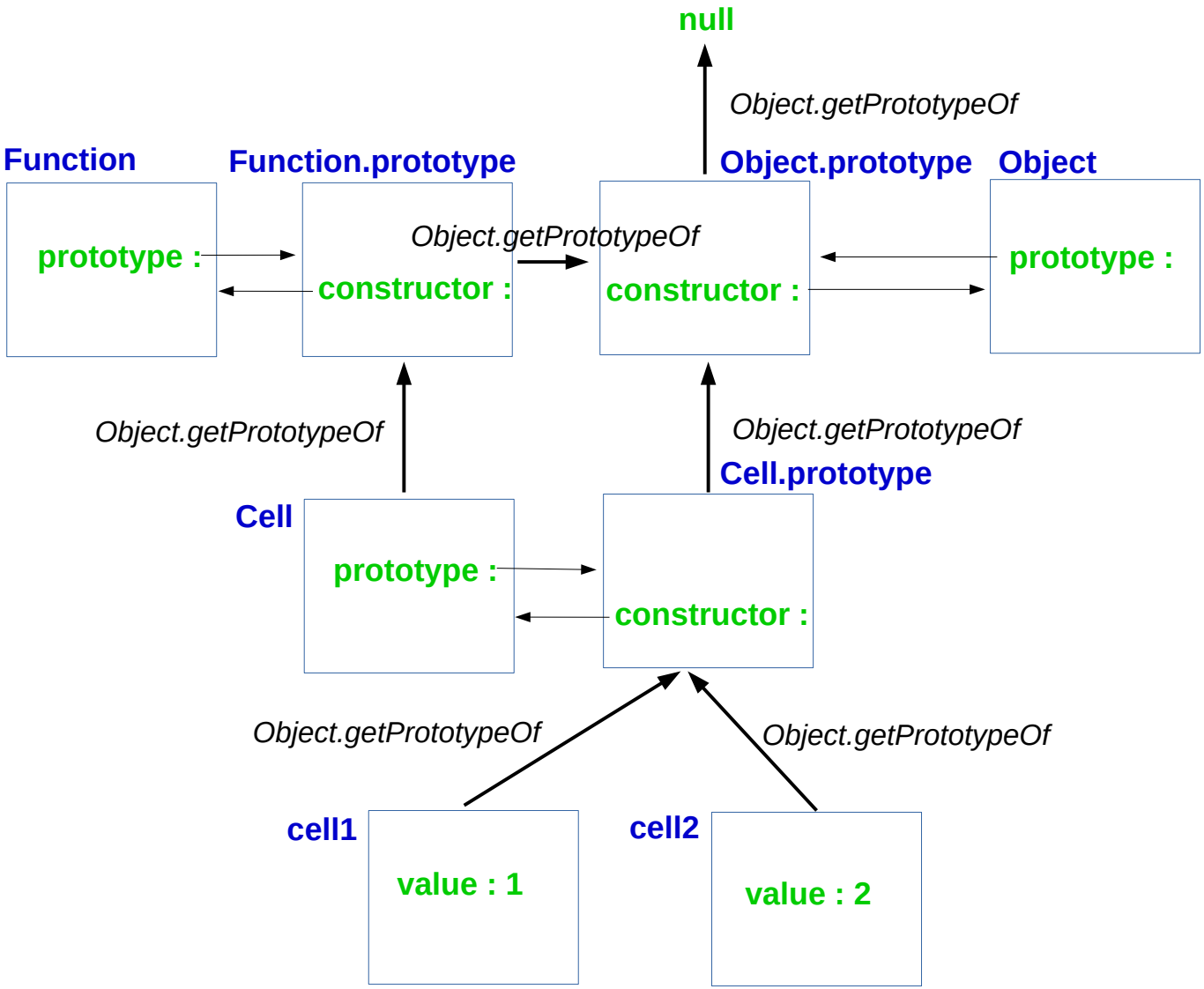
Example

```
> 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

```
function Shape(){} // works as an abstract class

// 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

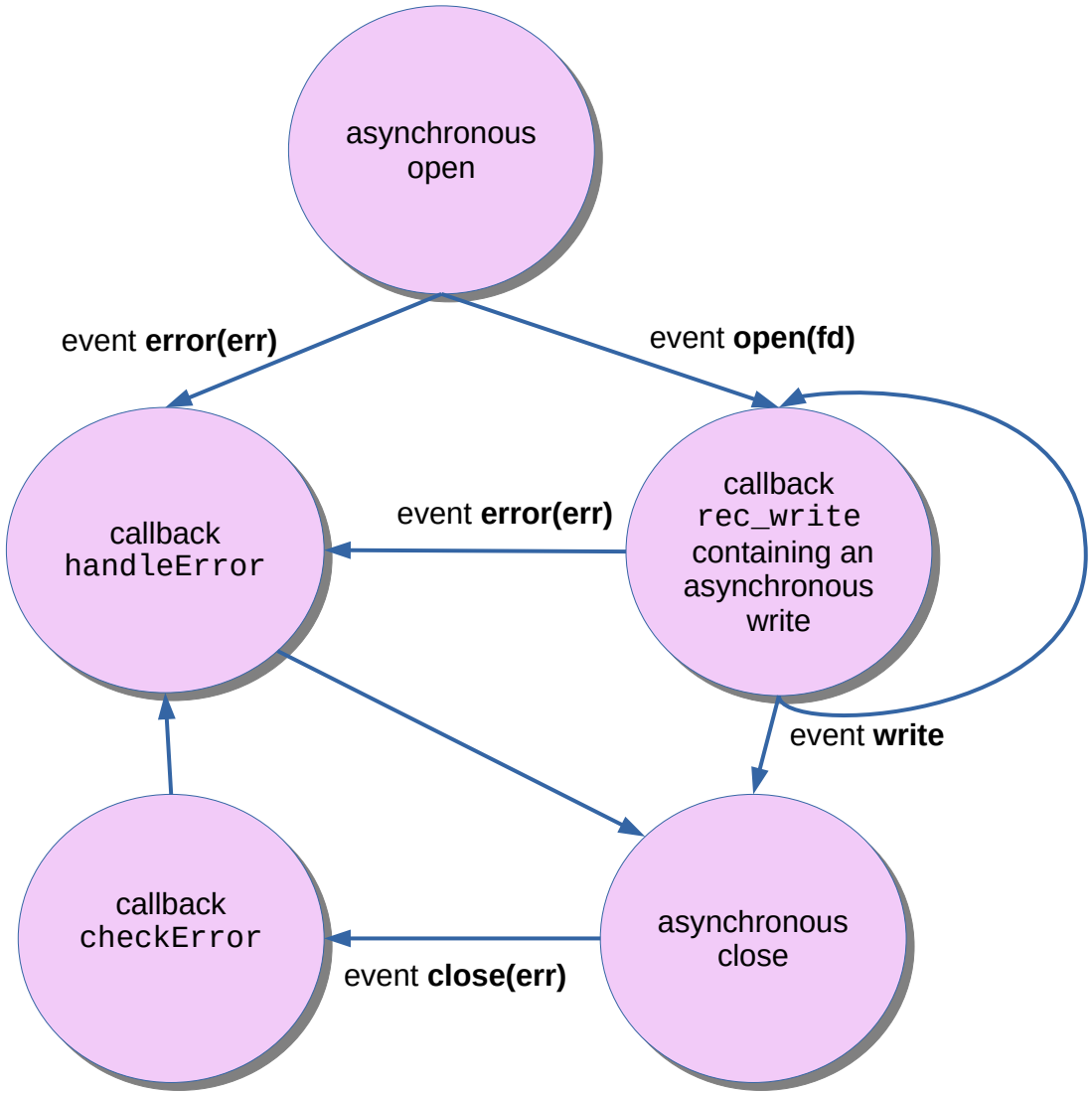
```
const { open, write, close } = require('fs'); // script fs_write.js

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 argument to write
    (function rec_write(err) {
        if (err) handleError(err, fd);
        else if (index < process.argv.length)
            write(fd, process.argv[index++] + '\n', rec_write);
        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

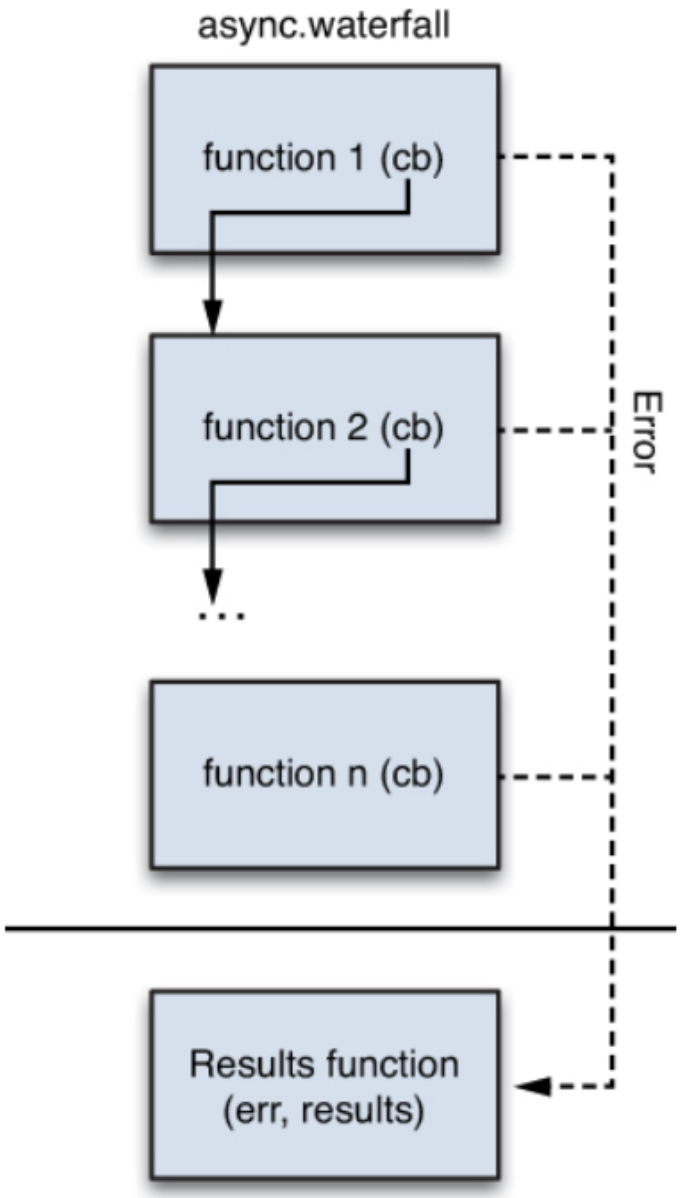
- items are iterated in order
- `cb`: hook to the next function to be called

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);
        });
});
```

Waterfall


async.waterfall
function 1 (cb)
function 2 (cb)
...
function n (cb)
Error
Results function
(err, results)

Waterfall

Example 1, with only synchronous functions

```
const { waterfall } = require('async'); // waterfall1.js
waterfall(
    [                                   // array of functions
        cb => cb(null, 'one', 'two'),   // function 1
        (arg1, arg2, cb) => {           // function 2
            console.log(arg1, arg2);
            cb(null, 'three');
        },
        (arg, cb) => {                  // function 3
            console.log(arg);
            cb(null, 'done');
        }
    ],
    (err, result) =>                    // results function
        err ? console.error(err.message) : console.log(result)
);
```

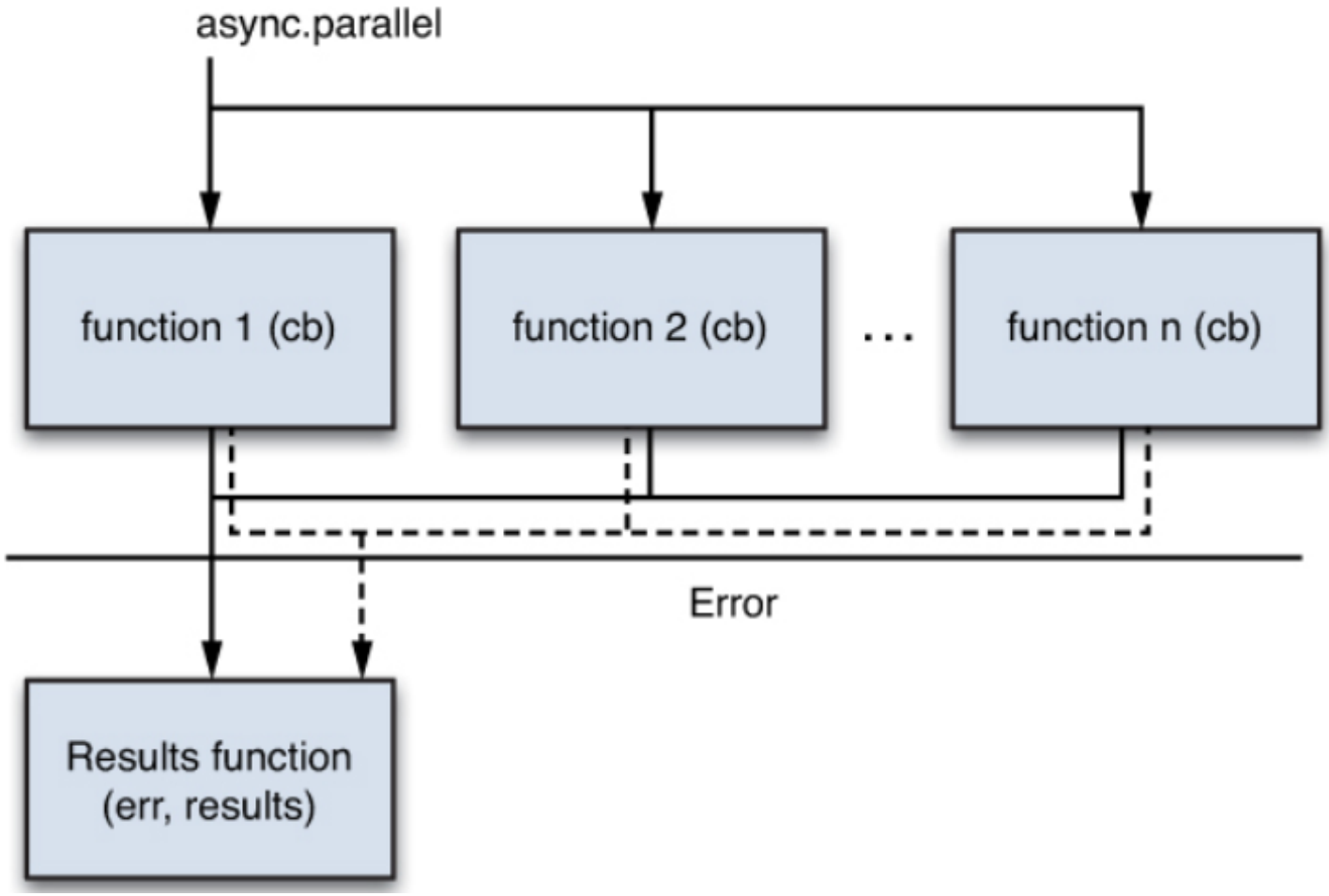
Remark: in case of errors, the last function is immediately called and the waterfall is interrupted
otherwise, the last function in the array passes the arguments to the last function

Waterfall

Example 2, with asynchronous functions

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

waterfall(
    [
        cb => cb(process.argv.length < 4 ? new Error('Missing args') : null),
        cb => readFile(process.argv[2], 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


async.parallel
function 1 (cb)
function 2 (cb)
...
function n (cb)
Error
Results function
(err, results)

Parallel

Example 1

```
const { parallel } = require('async'); // parallel1.js
parallel(
    [                                  // array of callbacks
        cb => {
            setTimeout(() => {
                console.log('Executing one');
                cb(null, 'one');
            }, 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(
    {                                  // object of callbacks
        one: cb => {
            setTimeout(() => {
                console.log('Executing one');
                cb(null, 'one', 'three');
            }, 200);
        },
        two: cb => {
            setTimeout(() => {
                console.log('Executing two');
                cb(null, 'two');
            }, 100);
        }
    },
    (err, res) => err ? console.error(err.message,res) : console.log(res)
);
```

Parallel

Example 3

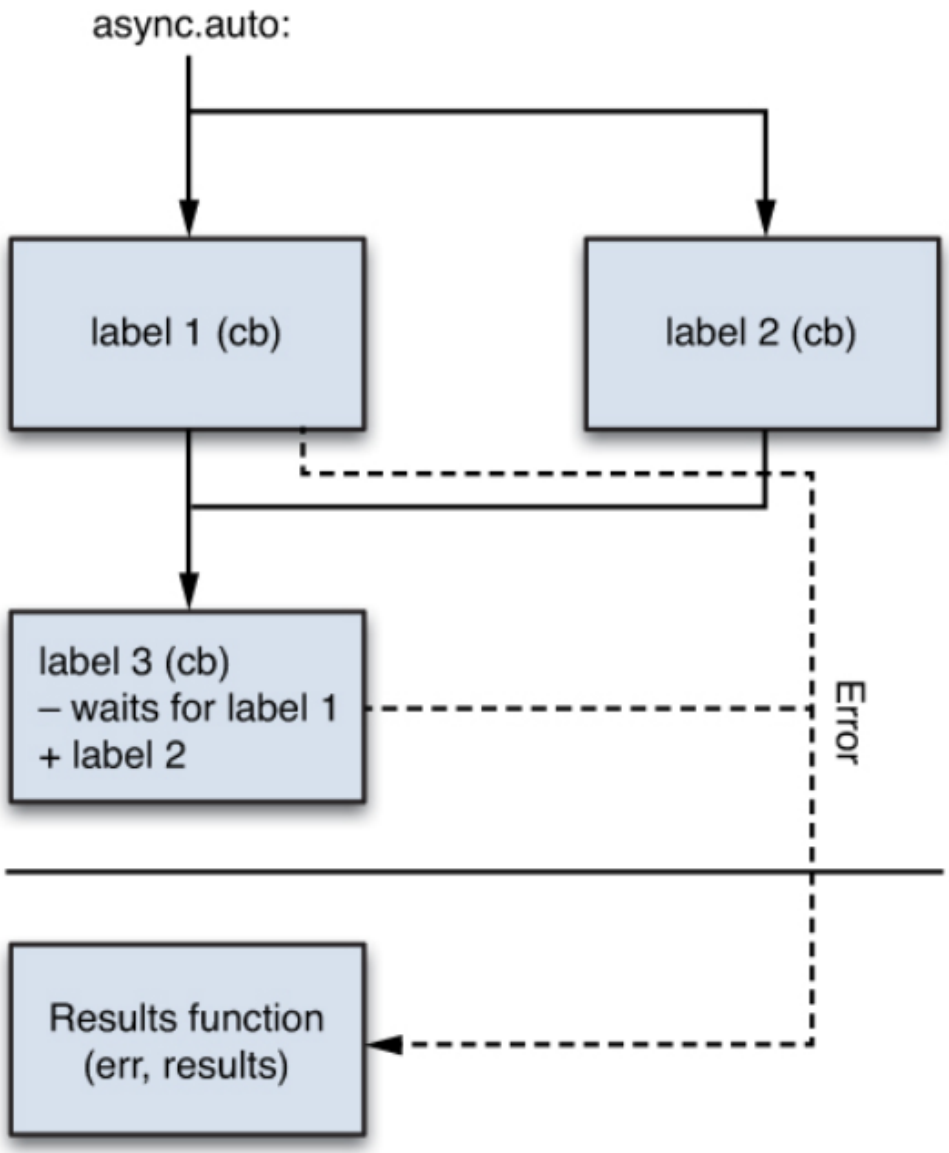
```
const { parallel } = require('async'); // parallel3.js
const { readFile } = require('fs');
const { request } = require('http');
parallel(
    {
        one: cb => {
            if (process.argv.length < 3)
                return cb(new Error('Missing file name'));
            readFile(process.argv[2],
                (err, data) => err ? cb(err) : httpRequest(data, 8080, cb));
        },
        two: cb => httpRequest('2', 8080, cb),
        three: cb => httpRequest('"three"', 8080, cb)
    },
    (err, res) => err ? console.error(err.message,res) : console.log(res)
);
```

Parallel

Example 3 (continued)

```
function httpRequest(postData, port, cb) {
    const options = {
        hostname: 'localhost',
        port: port,
        path: '/',
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
            'Content-Length': Buffer.byteLength(postData)
        }
    }
    const req = request(options, res => {
        const data = [];
        res.on('error', cb);
        res.on('data', chunk => data.push(chunk));
        res.on('end', () => cb(null, data.join('')));
    });
    req.on('error', cb);
    req.write(postData);
    req.end();
}
```

Auto


async.auto:
label 1 (cb)
label 2 (cb)
label 3 (cb)
– waits for label 1
+ label 2
Error
Results function
(err, results)

Auto

Example 1

```
const { readFile, writeFile } = require('fs'); // auto1.js
const { auto } = require('async');
auto(
    {                                                  // equivalent to a waterfall
        check: cb =>
            cb(process.argv.length < 4 ? new Error('Missing args') : null),
        read: ['check', (results, cb) => readFile(process.argv[2], 'utf8', cb)],
        write: ['read', (results, cb) => writeFile(process.argv[3], results.read,
      cb)]
    },
    (err, res) => err ? console.error(err.message,res) : console.log(res)
);
```

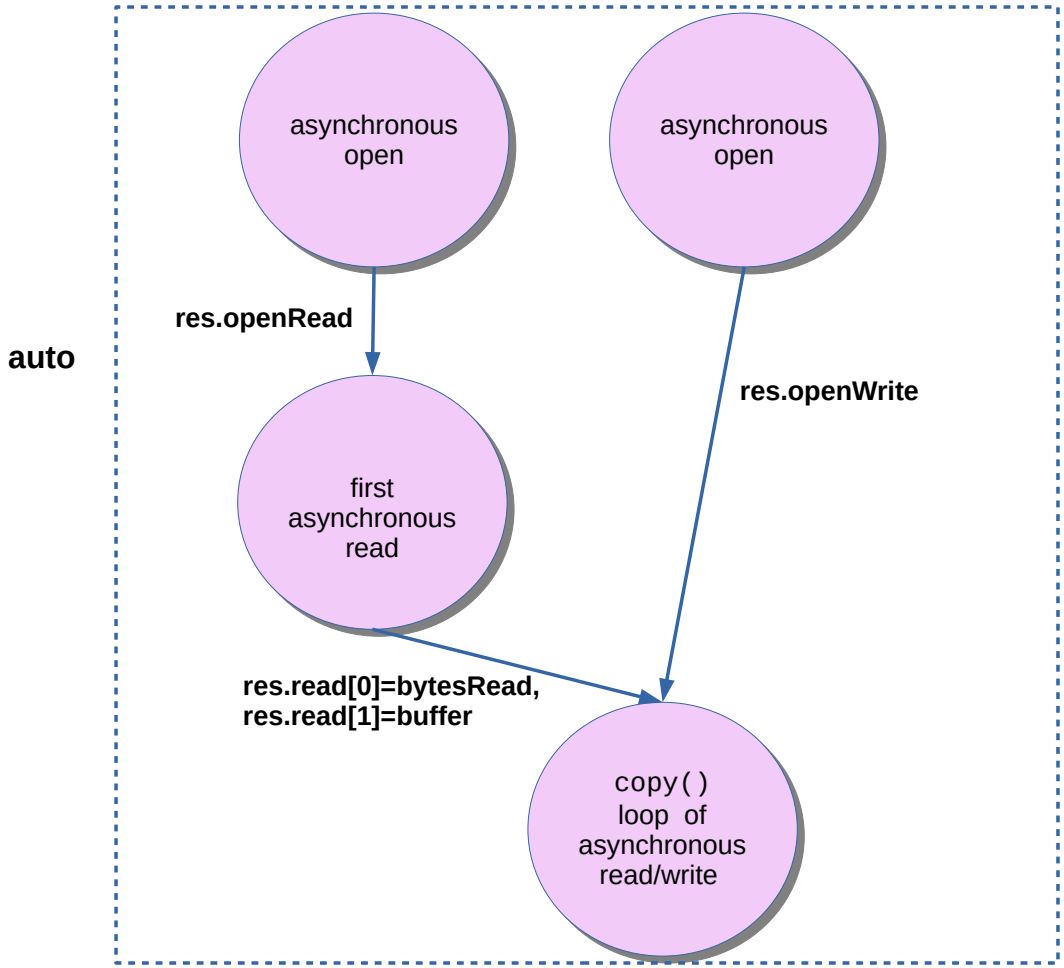
Auto

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)
);
```

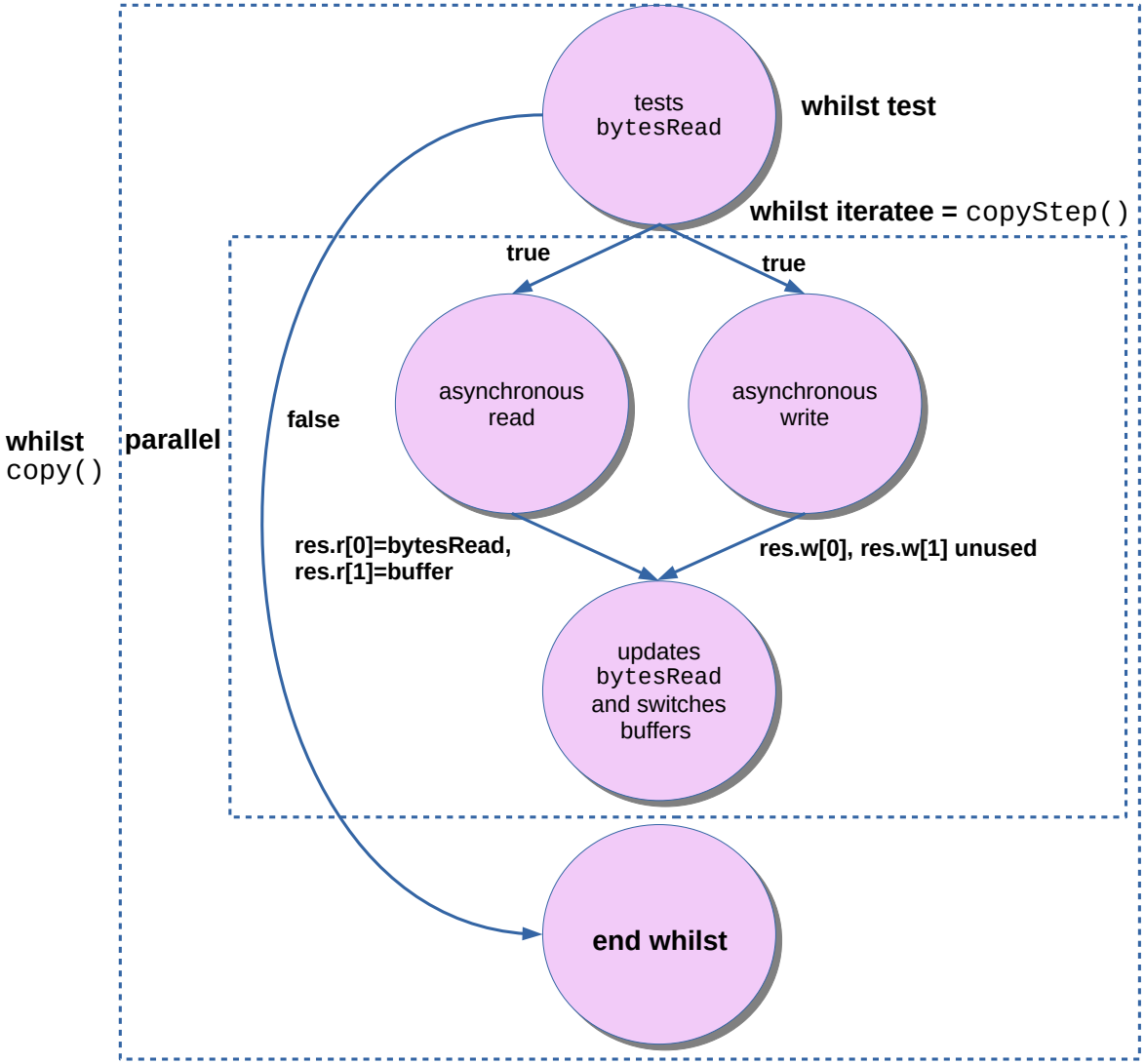
Auto

Example 3 (works with files of arbitrary size)



Auto

Example 3 (works with files of arbitrary size, continued)



Auto

Example 3 (works with files of arbitrary size, continued)

```
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); }
```

Auto

Example 3 (works with files of arbitrary size, continued)

```
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
        ...
    }
}
```

Auto

Example 3 (works with files of arbitrary size, continued)

```
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
}
```

Read streams

Type hierarchy

```
> const { createReadStream } = require("fs");
undefined;
> const reader = createReadStream('./in.txt');
undefined;
> reader.constructor;
[Function: ReadStream]
> Object.getPrototypeOf(reader);
[Function: Readable] {...}
> Object.getPrototypeOf(Object.getPrototypeOf(reader));
[Function: Stream] {...}
> Object.getPrototypeOf(Object.getPrototypeOf(Object.getPrototypeOf(reader)));
EventEmitter {...}
```

Read 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 => {
    console.log('data: ', chunk.toString());
    reader.pause();
    setTimeout(() => reader.resume(), 1000);
});

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'));
```

Read 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 => {
    console.log('data: ', chunk.toString());
    reader.pause();
    setTimeout(() => reader.resume(), 1000);
});

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'));
```

Write streams

Type hierarchy

```
> const { createWriteStream } = require("fs");
undefined;
> const writer = createWriteStream('./out.txt');
undefined;
> writer.constructor;
[Function: WriteStream]
> Object.getPrototypeOf(writer);
[Function: Writable] {...}
> Object.getPrototypeOf(Object.getPrototypeOf(writer));
[Function: Stream] {...}
> Object.getPrototypeOf(Object.getPrototypeOf(Object.getPrototypeOf(writer)));
EventEmitter {...}
```

Write streams

Example

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

writer.once('open', () => console.log('open'));
writer.once('ready', () => console.log('ready'));
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')) { // buffer limit exceeded
            console.log('pausing');
            setTimeout(write, 100, i + 1);
        }
        else
            write(i + 1);
    }
}

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

Write streams

Example with the 'drain' event instead of setTimeOut

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

writer.once('open', () => console.log('open'));
writer.once('ready', () => console.log('ready'));
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('pausing');
            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); });
```

Transform streams

In a nutshell

- Duplex streams relating the output to the input
- they implement both the Readable and Writable interfaces
- typical examples: compression/encryption of data streams

Transform streams

Example

converts 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) {
        this.push('\n'); // inserts a new line at the end
        callback();
    }
});

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

Transform streams

Another example

extracts city and temperature from JSON and coverts 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(); },
    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('Tranformation failed', err);
});
```

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

In a nutshell

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`: called with *v* if the promise is resolved with *v*
- `rej`: 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)`

- `resolve-reaction` called on *v* when *p* fulfilled with value *v*
- `reject-reaction` called on *err* when *p* rejected with error *err*

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 microtask queue
- the microtask queue has higher priority than the timer queue
- the microtask queue has lower priority than the `nextTick` 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);
```

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

`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)

Example

```
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 openRead() {
    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));
}

openRead();
```

async/await (ECMAScript 2017)

Example

```
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 openRead() {
  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); }
}

openRead();
```

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 openRead() {
  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); }
}

openRead();
```

async/await (ECMAScript 2017)

Rules

- the use of **`await`** allows automatic hadling 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)

Pros and cons

- simpler and more readable code with synchronous style
- more complex semantics, promotes less efficient synchronous programming

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, promotes less efficient synchronous programming

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, promotes less efficient synchronous programming

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/await (ECMAScript 2017)

Example with streams, pipelines and transforms

Without promises

```
const MAX = 21;
const { pipeline, Transform } = require('stream');
const data = require('./data.json'); // a JSON array

pipeline(
    data,
    new Transform({
        objectMode: true,
        transform(chunk, encoding, callback) {
            callback(null, chunk < MAX ? chunk + '\n' : undefined);
        }
    }),
    process.stdout,
    err => { if (err) console.error(err); });
```

async/await (ECMAScript 2017)

Example with streams, pipelines and transforms

With promises

```
const MAX = 21;
const { pipeline } = require('stream/promises');
const { Transform } = require('stream');
const data = require('./data.json'); // a JSON array

async function runPipe() {
    await pipeline(
        data,
        new Transform({
            objectMode: true,
            transform(chunk, encoding, callback) {
                callback(null, chunk < MAX ? chunk + '\n' : undefined);
            }
        }),
        process.stdout);
}

runPipe().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 file `package.json`
  example:

```
{
  "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
mod1.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:

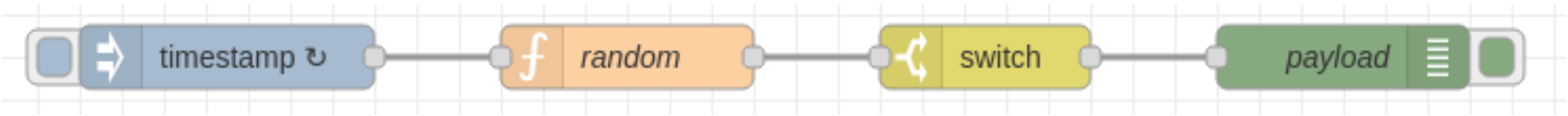
```
Point { x: 2, y: 2 } Point { x: 0, y: 0 }
```

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

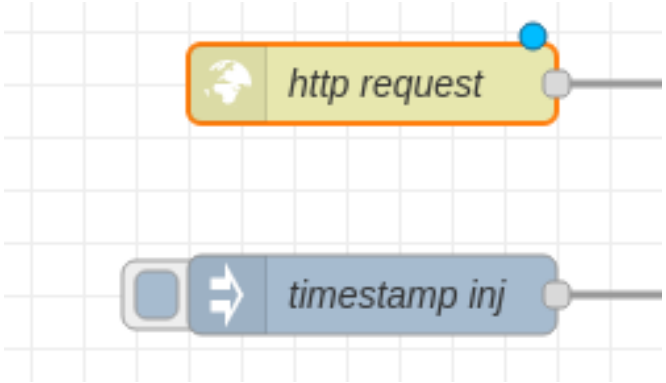


Flow-based programming

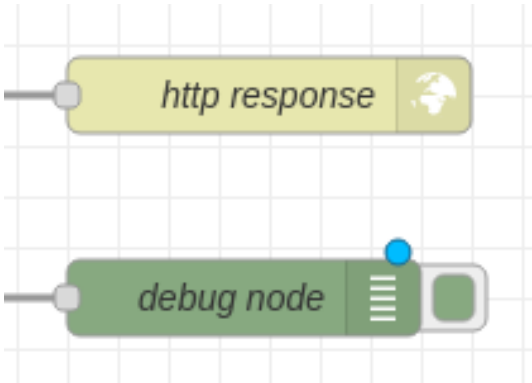
- programs are collections of flows
- flows are sequences of interconnected nodes where data flows and are processed
- nodes are the basic blocks of flows and perform simple operations on flowing data
  typically:
  - they have zero or one input, zero or more outputs
  - the initial node of the flow 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

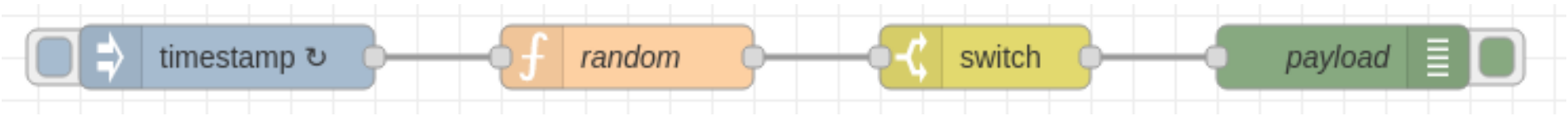
Example of initial nodes



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 demos