Foundations

Preparations

Tools

- Text editor
- Ability to run Javascript code
  - Recommended: node.js
  - Chrome
- Clone this repo:
  - `https://github.com/redradix/curso-javascript-pro`

redradix

Variable Declaration

Variable Declaration

- ES6 introduced the sentence **let**
    - fulfills the same function as **var**
    - it behaves slightly different

```
function myFunc() {
  console.log('value: ', x)
  var x = 12
  console.log('value: ', x)
}

myFunc()
```

redradix

```
function myFunc() {
  var x
  console.log('value: ', x)
  x = 12
  console.log('value: ', x)
}

myFunc()
```

```
function myFunc() {
  console.log('value: ', x)
  let x = 12
  console.log('value: ', x)
}

myFunc()
```

```
function myLoop() {
  for (var i=0; i <= 10; i++) {
    // no-op
  }
  return i
}
```

redradix

```
function myLetLoop() {
  for (let i=0; i <= 10; i++) {
    // no-op
  }
  return i
}
```

Exercise

- Find and fix the bug

```
function createFns() {
  let fns = []
  for (var i = 0; i< 10; i++) {
    fns.push(function() { console.log(i) })
  }
  return fns
}
```

Exercise

- Find and fix the bug

```
function randomNumber(n) {
  if (Math.random() > .5) {
    let base = 1
  } else {
    let base = -1
  }
  return base * n * Math.random()
}
```

```
function myFunc() {
  let a = 1
  let b = 0
  for (let i=4; i--;) {
    let b = a + 1
  }
  return b
}
```

```
function myFunc() {
  let a = 1
  for (let i=4; i--;) {
    let b = a + 1
  }
  return b
}
```

```
function myFunc() {
  let a = 1
  for (let i=4; i--;) {
    let a = a + 1
  }
  return a
}
```

```
function myFunc() {
  let a = 1
  for (let i=4; i--;) {
    let a = i + 1
  }
  return a
}
```

```
const one = 1
```

```
const one = 1;
one = 2; // ERROR! Assignment to constant variabe
```

Primitive Data Types

Data Types

- Javascript has **6** primitive data types

Data Types

- Javascript has **6** primitive data types
  - `Boolean`
  - `Number`
  - `String`
  - `Symbol`
  - `Null`
  - `Undefined`

Data Types

- **`typeof`** `operator`
  - Returns the name of the type of a given value

Data Types

```
typeof 42
```

Data Types

```
typeof "42"
```

Data Types

```
typeof undefined
```

redradix

Data Types

```
typeof null
```

String templates

```
const dynamic = 'interpolated value';
const final = `This is literal, this is ${dynamic}`;
console.log(final);
```

```
const dynamic = 'interpolated value';
const final = `This is literal, this is ${dynamic}`;
console.log(final);
```

redradix

```
const dynamic = 'interpolated value';
const final = `This is literal, this is ${dynamic}`;
console.log(final);
```

Exercise

- Using **string templates…**
    - Create a program that shows the time (*HH:MM:SS*) in the console *every second*

Exercise

- Using **string templates**...
  - Create a function that lists the elements of an array adding “**and**” between the last two
  - e.g.: **`[1, 2, 3] => “1, 2 and 3”`**

```
const user= {
  name: 'Elias',
  surname: 'Alonso'
}

console.log(`Welcome, ${user}`)
```

redradix

Exercise

- What can we **add** to the **user** object to show a better representation of its data when interpolated?
    - *tip: what does Javascript do to convert a value to a string?*

Symbols

Symbols

- First new data type since 1997
- Very specific function
- More or less similar to Lisp or Ruby symbols

Symbols

- Different from every other primitive data type
  - Don’t have any **literal representation**
  - Each symbol has a **unique value**
  - **Immutable**
  - **Not** converted to String automatically

redradix

Symbols

- Don’t have any **literal representation**
    - There is no syntax to represent their value
    - Only created through the factory function
    - Their value can’t be shown on the console

Symbols

```
const a = Symbol();

console.log(a); // Symbol()
```

Symbols

```
const a = Symbol('symbol a');

console.log(a); // Symbol(symbol a)
```

Symbols

- Each symbol has a **unique value**
  - All symbols are **different from each other**

Symbols

```
const a = Symbol();
const b = Symbol();

a === b;
```

Symbols

```
const a = Symbol();
const b = Symbol();

a === b; // false
```

Symbols

```
const a1 = Symbol('a');
const a2 = Symbol('a');

a1 === a2;
```

Symbols

```
const a1 = Symbol('a');
const a2 = Symbol('a');

a1 === a2; // false
```

Symbols

- **Not** converted to String automatically
  - Every other type is converted automatically to string

Symbols

```
const a = Symbol('a');
const str = a + '!';
```

Symbols

```
TypeError: Cannot convert a Symbol value to a string
```

Symbols

`Symbol([description])`

- Creates **a new (different) symbol each time**
- Can receive a description

Symbols

The language has a few **predefined symbols**

- `Symbol.iterator`
- `Symbol.hasInstance`
- `Symbol.match`

Symbols

*Symbols can be used as **property names***

redradix

Symbols

```
const p = Symbol('property');
const obj = {};
obj[p] = 'value';

console.log(obj[p]); // 'value'
```

redradix

Symbols

```
const p = Symbol('property');
const obj = {};
obj[p] = 'value';

console.log(obj[p]); // 'value'
```

redradix

Symbols

```
const p = Symbol('property');
const obj = {};
obj[p] = 'value';

p = null;
```

Symbols

```
const p = Symbol('property');
const obj = {};
obj[p] = 'value';

console.log(Object.keys(obj)); // []
```

Symbols

Symbols are useful for..

- Creating properties
- Inaccessible without the reference to the symbol

Symbols

Practical applications:

- Store **metadata**
- Store “private” info in **external objects**
- Configuration and special properties

Composite Data Types

Data Types

- Javascript has only **one** composite data type:

Data Types

- Javascript has only **one** composite data type:
  - **Object**

Data Types

What about ***arrays****?*

Data Types

```
typeof [1, 2]
```

Data Types

*What about **functions**?*

redradix

Data Types

`typeof console.log`

Data Types

"Functions are regular objects with the additional capability of being callable."

Fuente: [MDN]

redradix

Object

Object

- A dynamic set of properties
  - name: `string` or `symbol`
  - value: anything
- Can inherit properties from another object
- Handled by reference

Object

```
const obj = {};

const obj2 = { prop: 1 };

const obj3 = { ['a' + 'b']: 1 };
```

redradix

Object

```
const k = 'a';
const obj1 = { [k]: 1 };
const obj2 = { [k]: 1 };

obj1 === obj2; // ???
```

Object

```
const k = 'a';
const obj1 = { [k]: 1 };
const obj3 = obj1;

obj3.b = 2;

obj3 === obj1; // ???
```

Object.assign

Object

- `Object.assign`
    - Copies all properties from one object to another

```
const a = { a: 1 }
const b = { b: 2 }

Object.assign(a, b)

console.log(a)
console.log(b)
```

```
const a = { a: 1 }
const b = { b: 2 }
const c = { c: 3 }

Object.assign(a, b, c)
console.log(b)
```

```javascript
const a = { a: 1 }
const b = { b: 2 }
const c = { c: 3 }
const x = Object.assign(a, b, c)

console.log(x) // { a: 1, b: 2, c: 3 };
```

```
const a = { a: 1 }
const b = { b: 2 }
const c = { c: 3 }

const x = Object.assign(a, b, c)

x === a // true
```

Exercise

- How can we merge **a**, **b** and **c** without modifying any of the three?

Exercise

- Write a function **clone** that receives an object as the first parameter and returns **a copy**

```javascript
const u1 = { username: 'root', password: 'iamgod' }
const u2 = { username: 'luser', password: '12345' }
const users = { u1: u1, u2: u2 }

const usersCopy = clone(users);
usersCopy.u3 = { username: 'admin', password: 'aDS00Dkxx098Sd' }

console.log(users.u3) // ???

usersCopy.u1.username = 'p0wnd'

console.log(users.u1.username) // ???

users.u1 === usersCopy.u1 // ???
```

redradix

Exercise

- Enhance **clone** to avoid the hack shown in the previous example
    - perform a **deep copy**

```
const u1 = { a: { b: { c: 1 } } }
const u2 = { a: { b: { d: 2 } } }

const x = Object.assign({}, u1, u2)
console.log(x.a.b) // ???
```

Exercise

- Write **merge**, the recursive version of **Object.assign**

```
const u1 = { a: { b: { c: 1 } } }
const u2 = { a: { b: { d: 2 } } }

const x = merge({}, u1, u2)
console.log(x.a.b) // { c: 1, d: 2 }
```

```
function merge(base, ...args) {
  Object.assign(base, ...args)
  for (let [key, value] of Object.entries(base))
    if (value instanceof Object)
      base[key] = merge(value, ...args.map(function(arg) {
        return (arg[key] || {})
      }))
  return base
}
```

```
const u1 = { a: { b: { c: 1 } }, b: 3, c: 4 }
const u2 = { a: { b: { d: 2 } }, b: 2 }
const u3 = { x: 3, a: { c: 'hello' } }

const x = merge(u1, u2, u3)
console.log(x)
console.log(u1)
console.log(u2)
```

```
const config = {
  server: {
    hostname: 'myapp.domain.com',
    port: 443,
    protocol: 'https'
  },
  database: {
    host: '192.169.1.2',
    port: 33299
  }
}

const testConfig = merge(config, {
  server: { hostname: 'localhost' },
  database: { host: 'localhost' }
})
```

```
const x = [{ a: 1 }, [{ b: 2 }]]
const y = [{ b: 2 }, [], { c: 'hi' }]

console.log(merge(x, y))
```

Object.defineProperty

Object.defineProperty

- **Object.defineProperty**
  - **configure** the properties of an object
    - modify its **value**
    - manage if it is **enumerable**
    - control if it is **read-only**
    - decide if it can be **reconfigured**

```js
const obj = {}
Object.defineProperty(obj, 'a', {
  value: 1
})

console.log(obj.a) // 1
```

```js
const obj = {}
Object.defineProperties(obj, {
  b: { value: 2 },
  c: { value: 3 }
})

console.log(obj.b) // 2
console.log(obj.c) // 3
```

redradix

```
const obj = {}
Object.defineProperties(obj, {
  b: { value: 2 },
  c: { value: 3 }
})

console.log(obj) // ????
```

```javascript
const obj = {}
Object.defineProperties(obj, {
  b: { value: 2 },
  c: { value: 3 }
})

console.log(obj) // {}
```

```
const obj = {}
Object.defineProperties(obj, {
  b: { value: 2 },
  c: { value: 3 }
})

console.log(Object.keys(obj)) // []
```

```
const obj = {}
Object.defineProperties(obj, {
  b: { value: 2, enumerable: true },
  c: { value: 3, enumerable: true }
})

console.log(obj) // { b: 2, c: 3 }
```

```
const obj = {}

Object.defineProperty(obj, 'a', { value: 1 })

Object.defineProperty(obj, 'a', {
  value: 2,
  enumerable: true
})
```

`TypeError: Cannot redefine property: a`

```js
const obj = {}

Object.defineProperty(obj, 'a', {
  value: 1,
  configurable: true
})

Object.defineProperty(obj, 'a', {
  value: 2,
  enumerable: true
})
```

Object

- Property descriptor:
  - `value` (*undefined*)
  - `enumerable` (*false*)
  - `configurable` (*false*)
  - `writable` (*false*)

getters and setters

Object

- The property descriptor can also define:
  - `get`
  - `set`

redradix

Object

```
const obj = {};
Object.defineProperty(obj, 'random', {
  get: function() {
    console.log('Throwing the dice...');
    return Math.floor(Math.random() * 100);
  }
});
console.log(obj.random); // Throwing the dice... 27
console.log(obj.random); // Throwing the dice... 18
```

Object

```
const obj = {};

Object.defineProperty(obj, 'a', {
  get: function() {
    return this.a * 2;
  }
});

obj.a = 2;
console.log(obj.a); // ???
```

redradix

Object

```
const temp = { celsius: 0 };

Object.defineProperty(temp, 'fahrenheit', {
  set: function(value) {
    this.celsius = (value - 32) * 5/9;
  },
  get: function() {
    return this.celsius * 9/5 + 32;
  }
});
```

Object

```
temp.fahrenheit = 10;
console.log(temp.celsius); // -12.22

temp.celsius = 30;
console.log(temp.fahrenheit); // 86
```

Object

```
const obj = {};
obj.fahrenheit = temp.fahrenheit; // 86

obj.celsius = -12.22;
console.log(obj.fahrenheit); // ???
```

Exercise

- Write `withAccessCount(object, propertyName)`
  - a **function**
  - receives an **object** and the **name of a property**
  - **counts** how many times that property **has been accessed**
  - adds the method **getAccessCount** to the object

```
const obj = { p: 1 }
withAccessCount(obj, 'p')

obj.p = 12
console.log(obj.p)
console.log(obj.p)

console.log(obj.getAccessCount('p')) // 2
```

```
const obj = { p: 1, j: 2 }
withAccessCount(obj, 'p')
withAccessCount(obj, 'j')

console.log(obj.p)
console.log(obj.p)
console.log(obj.j)

console.log('->', obj.getAccessCount('p')) // 2
console.log('->', obj.getAccessCount('j')) // 1
```

Object

```
const obj = {
  get prop() {
    return this._value
  },
  set prop(value) {
    this._value = value * 2
  }
}
```

redradix

Exercise

- Write `withDynamicAverage(array)`
    - a **function**
    - receives an array
    - adds a calculated property **average** that returns the average of all the values in the array

Seal objects

redradix

Object

`Object.seal(obj)`

- Seals an object
    - Prevents creation of new properties
    - Prevents deletion of existing properties
    - All existing properties become not configurable

Object

```
const obj = { a: 1, b: 2, c: 3 };

Object.seal(obj);

obj.c = 0;
obj.d = 4;
console.log(obj); // { a: 1, b: 2, c: 0 }

delete obj.a;
console.log(obj);  // { a: 1, b: 2, c: 0 }
```

Object

```
Object.freeze(obj)
```

- Makes the object immutable
  - Prevents creation of new properties
  - Prevents deletion of existing properties
  - Values of existing properties can't be changed

Object

```
const obj = { a: 1, b: 2, c: 3 };

Object.freeze(obj);

obj.c = 0;
obj.d = 4;
delete obj.a;
console.log(obj); // { a: 1, b: 2, c: 3 }
```

redradix

Object.create

Object

`Object.create(proto, properties)`

- Creates a new object
  - **`proto`**: object prototype
  - **`properties`**: property descriptors

redradix

Object

```
const obj = { a: 1, b: 2 };
console.log(obj); // { a: 1, b: 2 }
console.log(obj.toString); // ???
```

Object

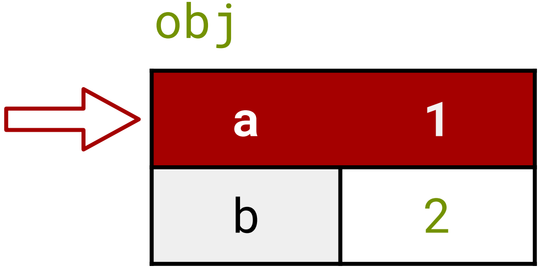
```
const obj = { a: 1, b: 2 };
```

obj

a	1
b	2

Object

```
obj.a // 1
```



Object

```
obj.toString // [Function: toString]
```

obj

a	1
b	2

???

Object

```
obj.toString // [Function: toString]
```

obj

a	1
b	2
proto	Object

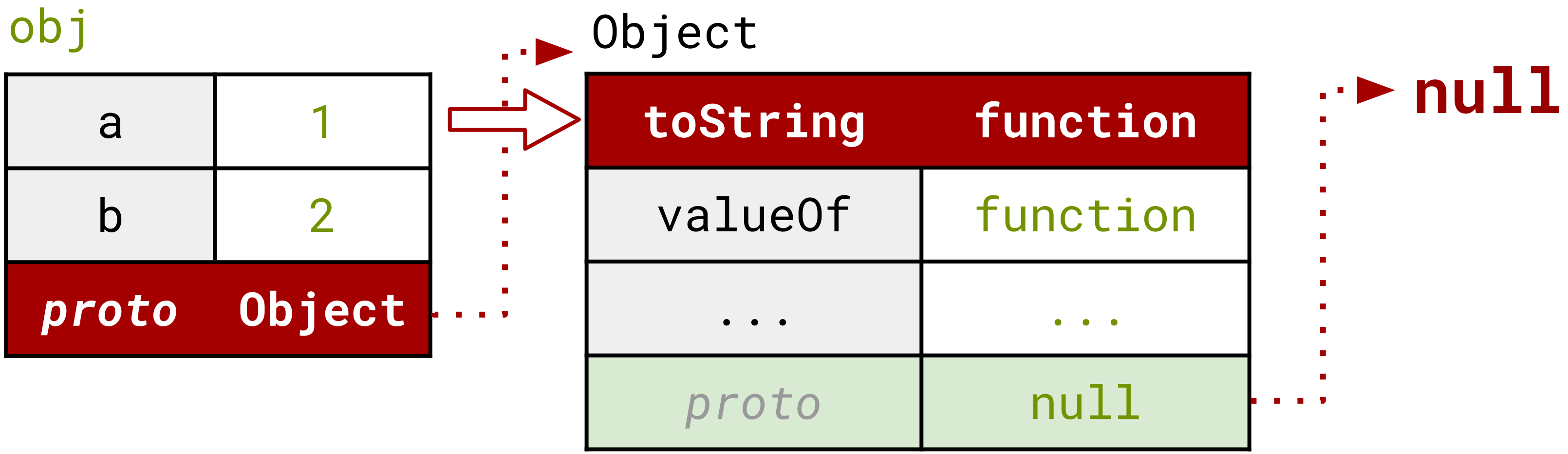
Object

toString	function
valueOf	function
...	...
proto	null

null

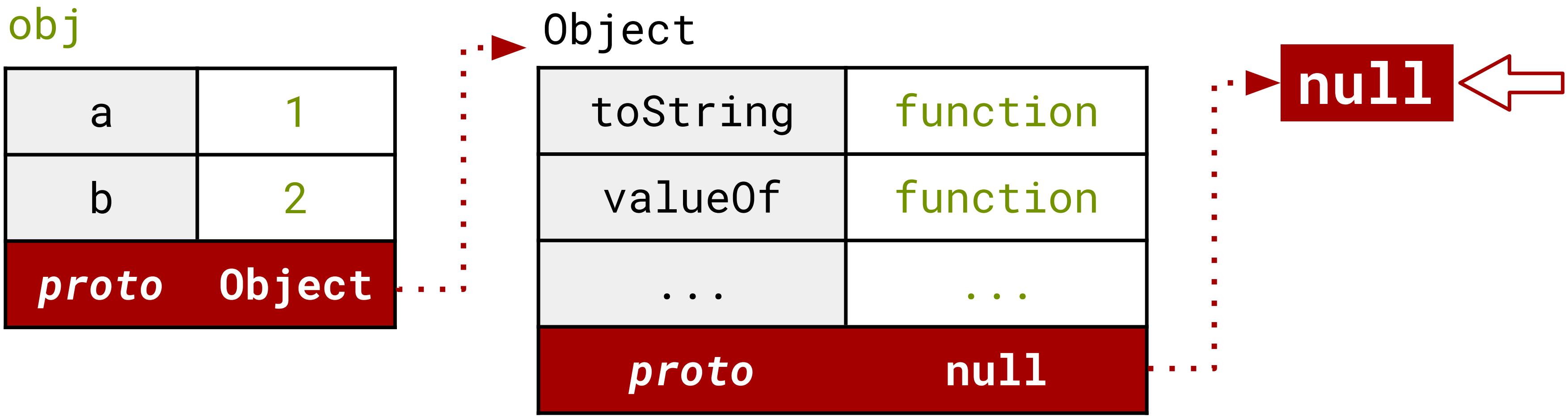
Object

```
obj.toString // [Function: toString]
```



Object

```
obj.notFound // undefined
```

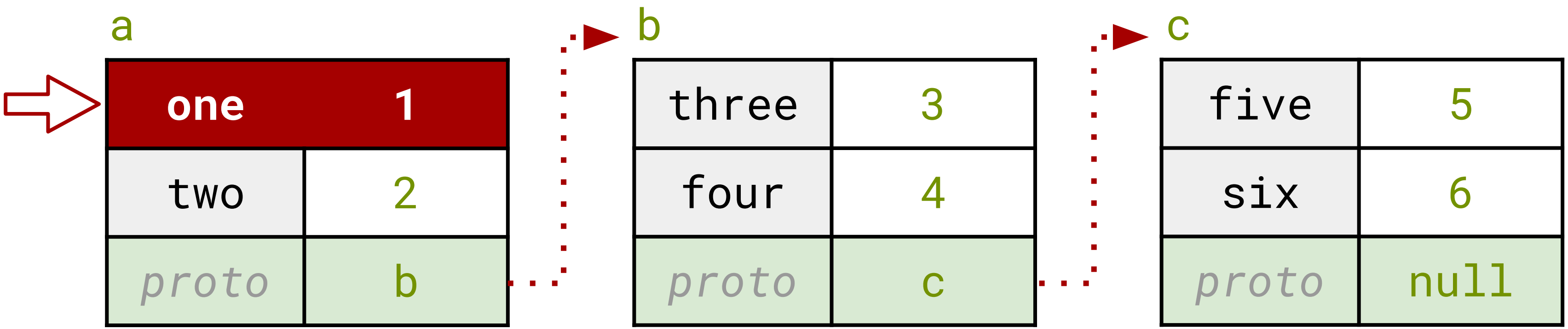


Object

- If **A** is the prototype of **B**...
  - All properties of **A** are visible on **B**
  - All properties of the prototype of **A** are visible on **B**
  - All properties of the prototype of the prototype of **A** are visible on **B**
  - ....

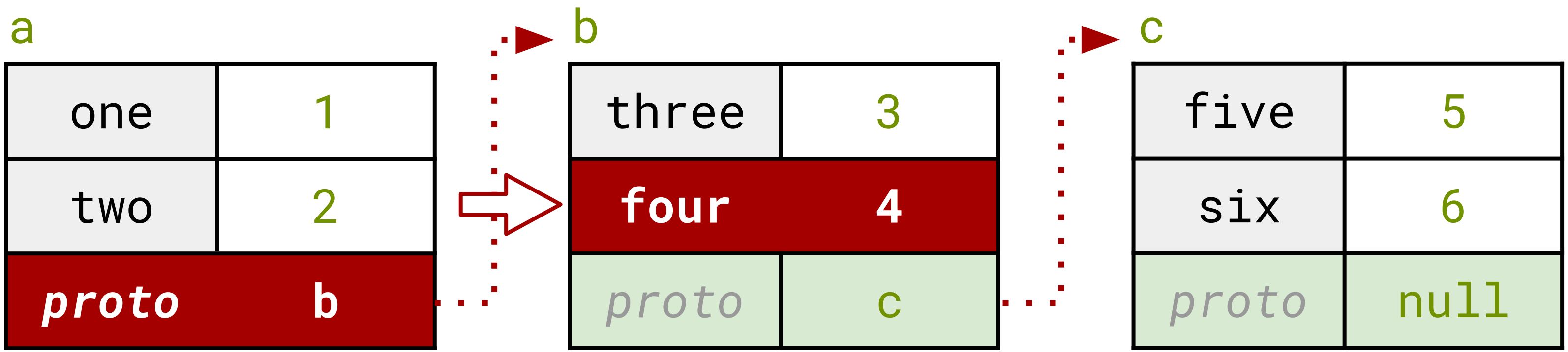
Object

```
a.one // 1
```



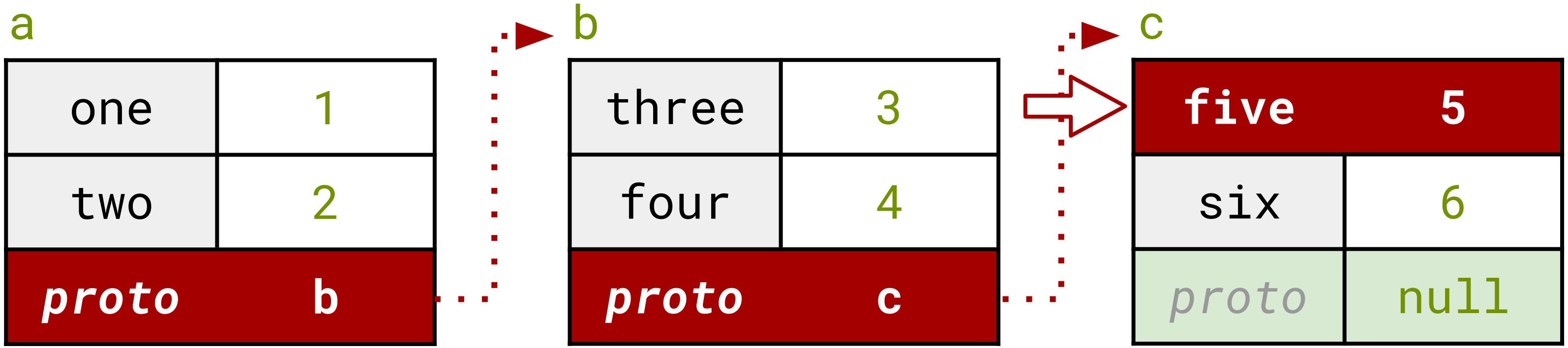
Object

```
a.four // 4
```



Object

```
a.five // 5
```



Exercise

- Build an object **C** with **null** as its prototype
- Build an object **B** with **C** as its prototype
- Build an object **A** with **B** as its prototype
  - Like in the previous example

Exercise

- What is the result of this call: **`a.toString()`** ?
- Why?

Object

`obj.hasOwnProperty(prop)`

- Tests if the property **prop** belongs to the object **obj**
- Useful to differentiate between own and inherited properties

Object

```
const obj = Object.create({ a: 1 }, {
  b: { value: 2 },
  c: { value: 3, enumerable: true }
});

obj.hasOwnProperty('a'); // false
obj.hasOwnProperty('b'); // true
obj.hasOwnProperty('c'); // true
```

redradix

Object

```
const base = { common: 'one' };

const a = Object.create(base, {
  name: { value: 'a' }
});

a.name; // 'a'
a.common; // ???
```

Object

```
base.common = 'two';

const b = Object.create(base, {
  name: { value: 'b' }
});

b.name; // 'b'
b.common; // ???
```

Object

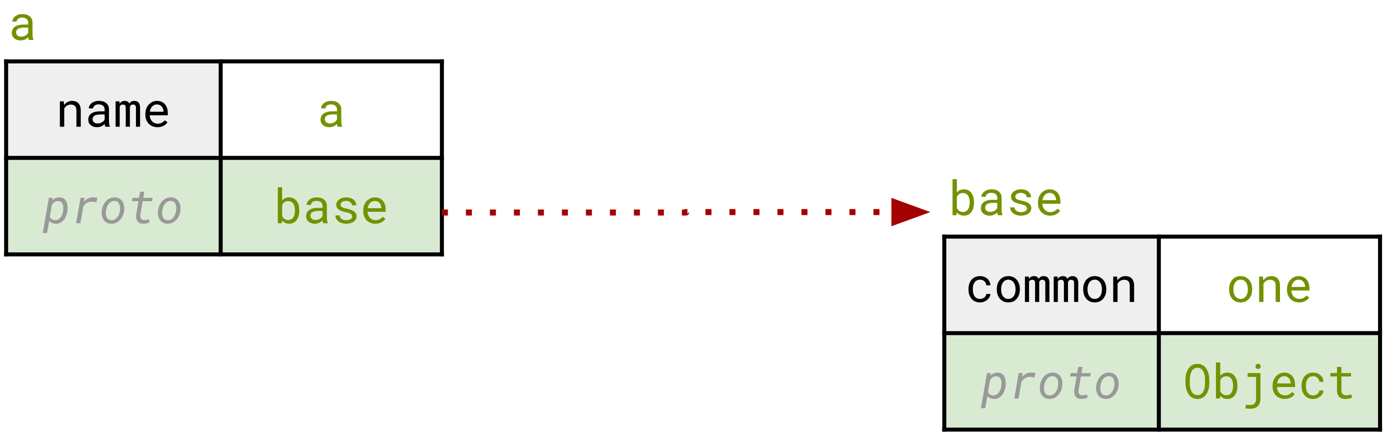
```
a.common === b.common; // ???
```

redradix

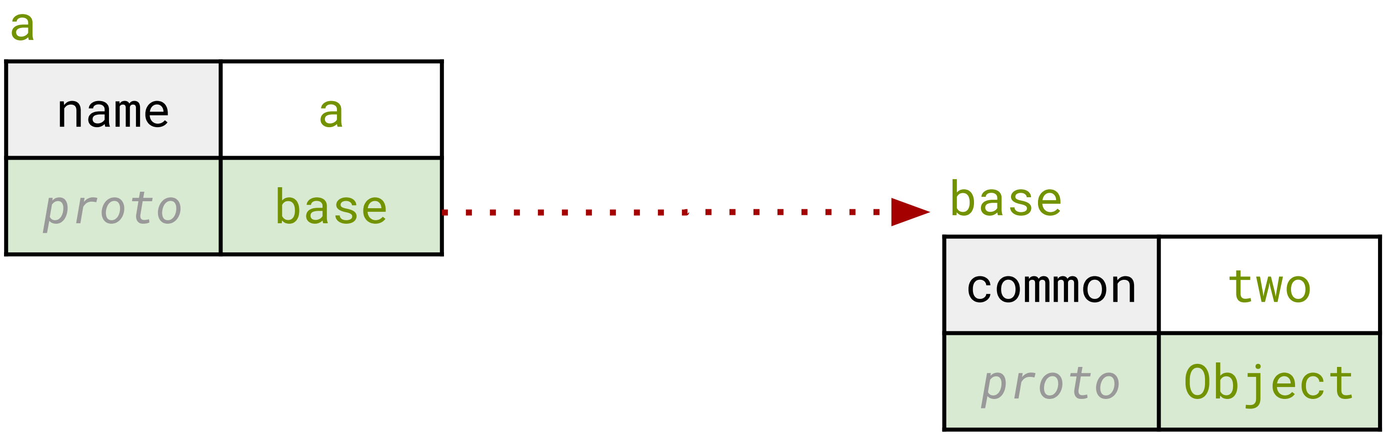
Object

```
a.common; // ???
```

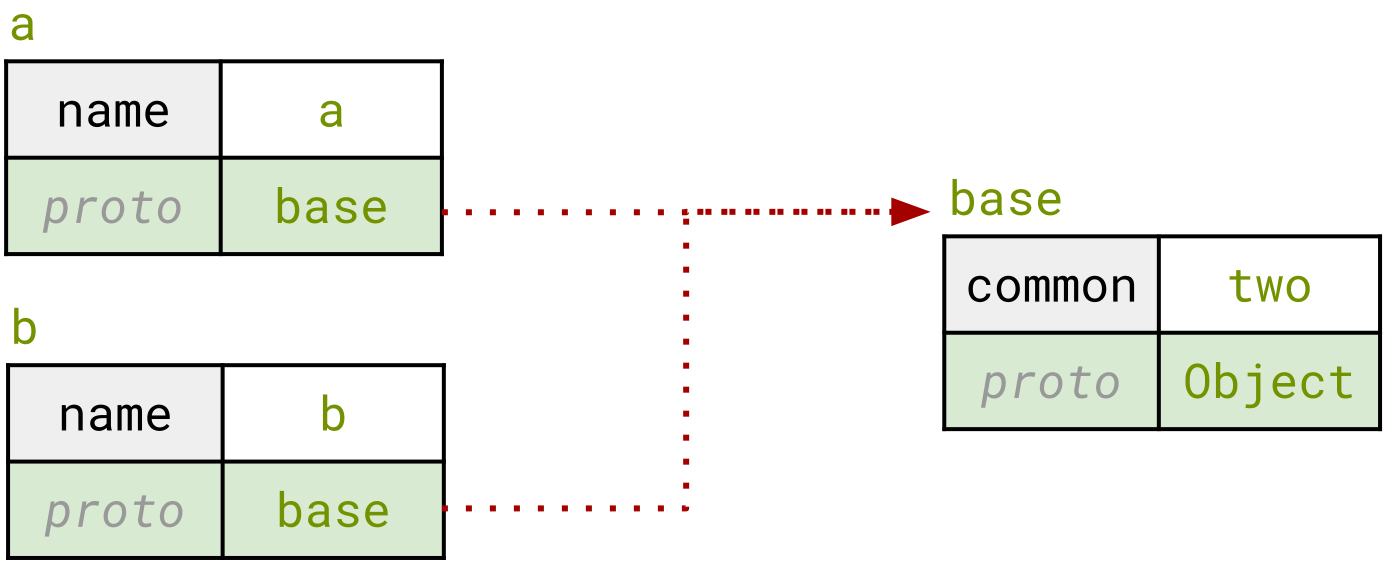
Object



Object



Object


a
name
a
proto
base
b
name
b
proto
base
base
common
two
proto
Object

Object

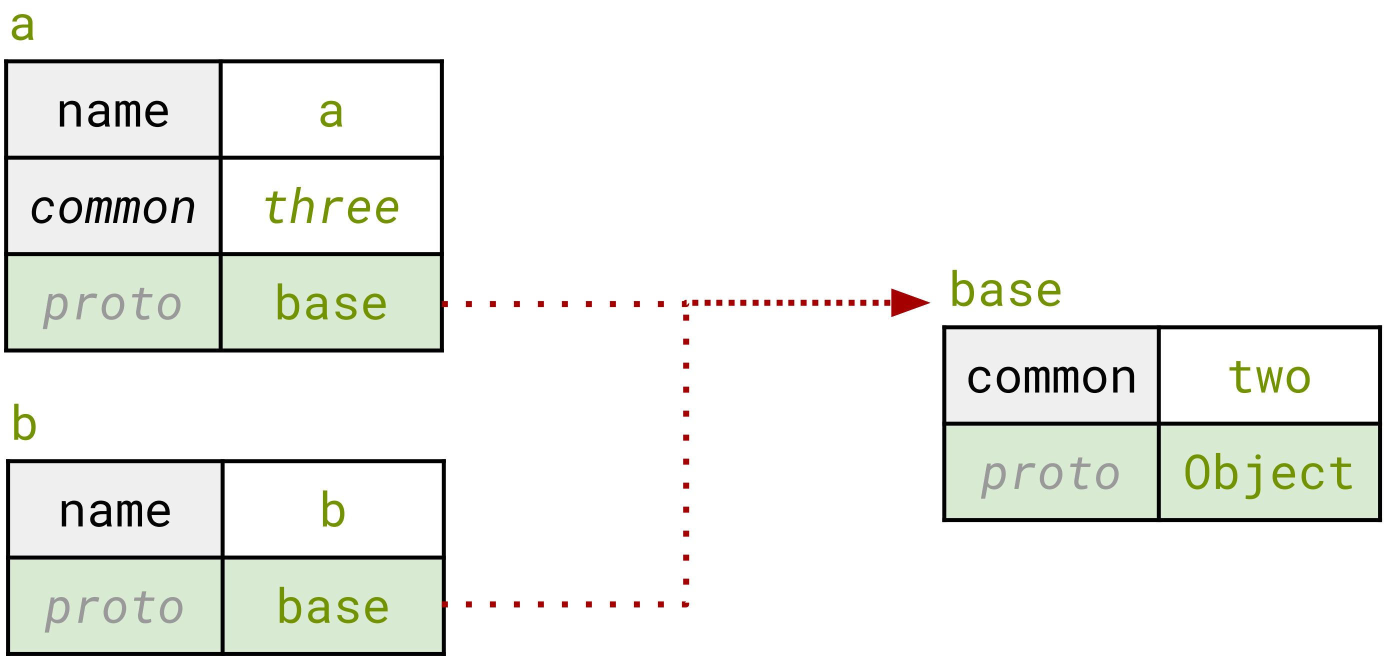
```
a.common = 'three';
b.common; // ???
```

Object

```
a.common === b.common; // ???
```

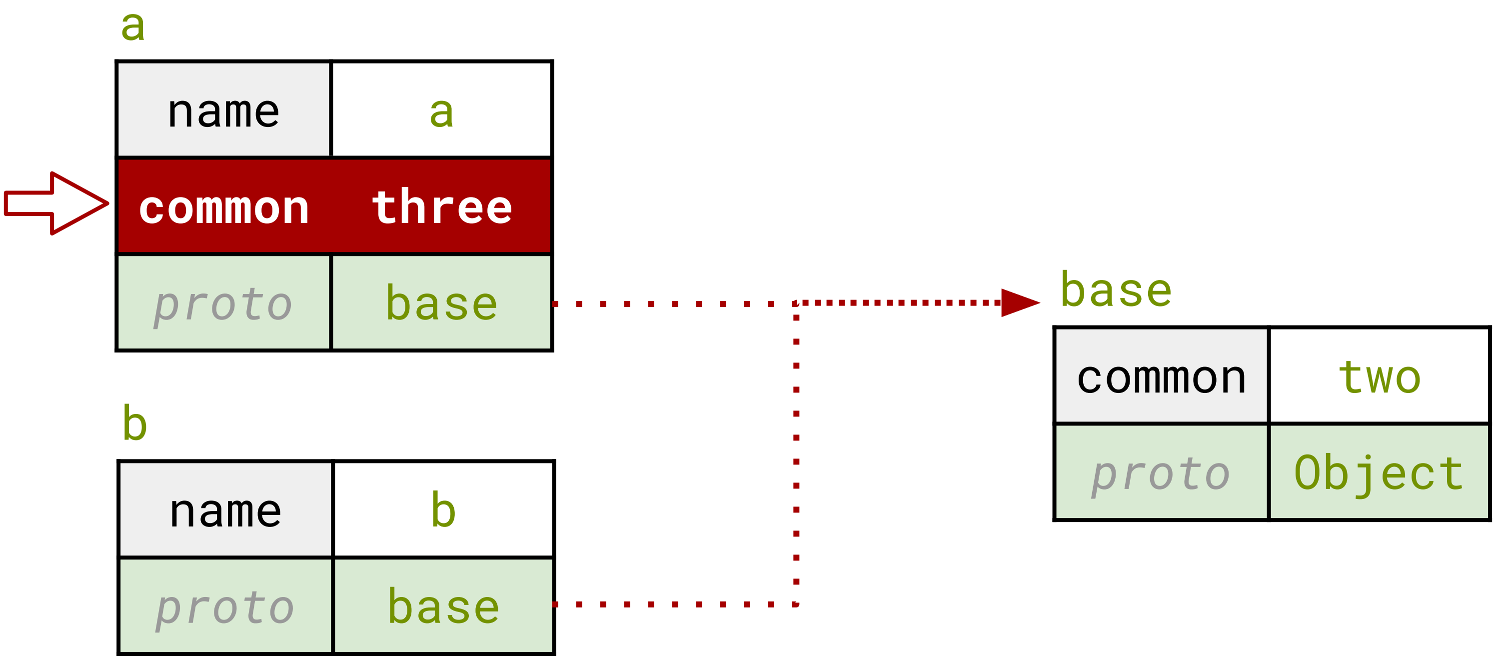
Object

```
a.common = 'three';
```



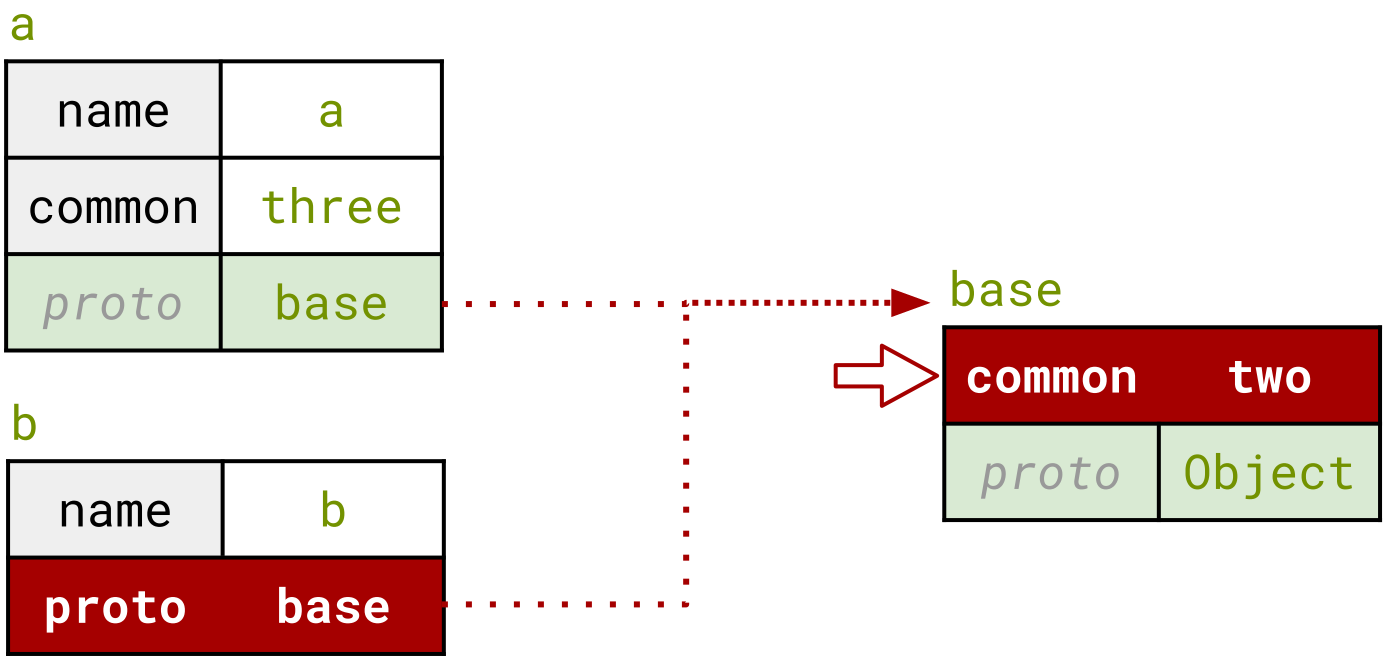
Object

a.common



Object

b.common



Object

- The prototype chain is an **asymmetric** mechanism:
  - **reads** propagate up the chain
  - **writes** don't
- Suitable for sharing a set of common properties among many children
  - and store in each children only its differences

Object

```
const list = {
  items: [],
  add: function(el) { this.items.push(el); },
  getItems: function() { return this.items; }
};
```

Object

```
const todo = Object.create(list);

todo.add('Write some tests');
todo.add('Refactor code');
todo.add('Run the tests');

todo.getItems(); // ???
```

Object

```
const shopping = Object.create(list);

shopping.add('Eggs');
shopping.add('Ham');
shopping.add('Milk');

shopping.getItems(); // ???
```

redradix

Object

But... ***Why?***

Object

```
const todo = Object.create(list);
```

Object

```
this.items.push(el);
```

Object

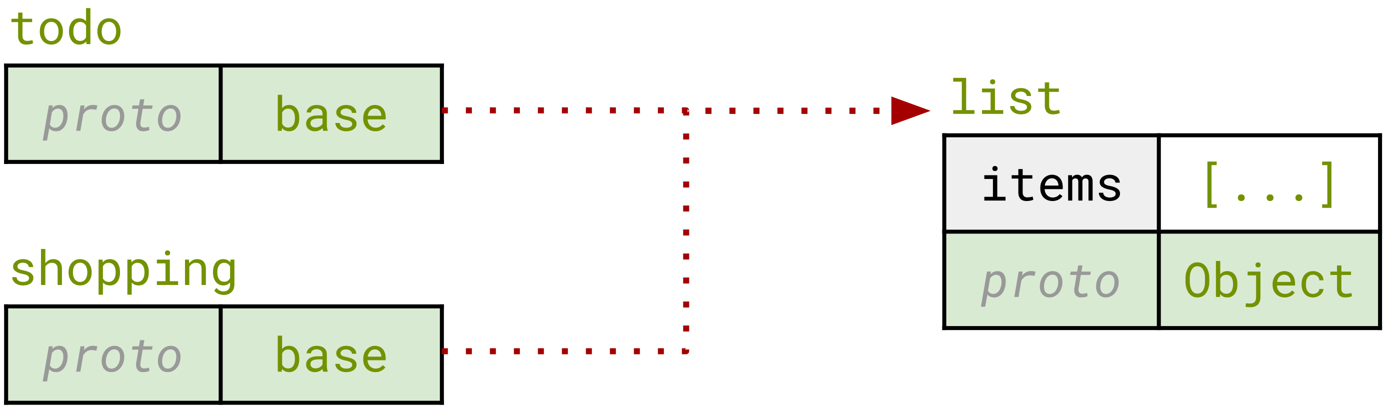
```
this.items.push(el);
```

Object

```
this.items.push(el);
```

Object

```
const shopping = Object.create(list);
```



Object

```
const parent = Object.create(null, {
  x: { writable: false, value: 1 }
});

const child = Object.create(parent);

child.x = 2;

child.x; // ???
```

redradix

Object

```
const parent = Object.create({}, {
  km: { value: 0, writable: true },
  mi: {
    get: function() { return this.km / 1.60934; },
    set: function(v) { this.km = v * 1.60934; }
  }
});
```

Object

```
const child = Object.create(parent);
child.mi = 80;

child.km; // ???
parent.km; // ???
```

redradix

Functions

Receiver

Given:

```
const obj = {
  name: 'Homer',
  greet: () => {
    console.log(`Hi, ${this.nombre}`)
  }
};
```

What’s the meaning of this?

```
obj.name;
```

Receiver

Given:

```
const obj = {
  name: 'Homer',
  greet: () => {
    console.log(`Hi, ${this.nombre}`)
  }
};
```

and this?

```
obj.greet;
```

Receiver

Given:

```
const obj = {
  name: 'Homer',
  greet: () => {
    console.log(`Hi, ${this.nombre}`)
  }
};
```

and this?

```
obj.greet();
```

Receiver

Given:

```
const obj = {
  name: 'Homer',
  greet: () => {
    console.log(`Hi, ${this.nombre}`)
  }
};
```

Is this the same?

```
const greet = obj.greet;
greet();
```

NO

Receiver

```
obj.greet();
```

```
const greet = obj.greet;
greet();
```

1. ***Send the message*** "*greet*" to obj
2. ***obj handles the execution*** of the associated function
3. obj is the ***receiver***

1. ***Access to the value of the property*** "*greet*" of obj
2. I assume it is a function and ***invoke it***
3. **There is NO receiver**

Receiver

There are four ways to invoke a function:

1. **Direct invocation**

Receiver

There are four ways to invoke a function:

1. Direct invocation
2. **Sending a message to an object (method)**

Receiver

- The receiver of the method...
  - A reference **without lexican binding**
  - Holds a reference to the object **that receives the invocation**
    - *Whatever is at the left side of the invocation dot*
  - Gets **its value** at **invocation time**

Receiver

`this`

redradix

```javascript
const obj = {
  counter: 0,
  increment: function() {
    this.counter++
  }
}

obj.increment()

console.log(obj.counter)
```

```
const obj = {
  counter: 0,
  increment: function() {
    this.counter++
  }
}

obj.increment()

console.log(obj.counter)
```

```
const obj = {
  counter: 0,
  increment: function() {
    this.counter++
  }
}

obj.increment()

console.log(obj.counter)
```

redradix

```
const obj = {
  counter: 0,
  increment: function() {
    this.counter++
  }
}

obj.increment()

console.log(obj.counter)
```

Receiver

- Invoking a method **its not just** calling a function
    - There is a **receiver**
        - Whatever is at the left of the **invocation dot**
    - Additional steps
        - **bind `this`**

```
const obj = {
  counter: 0,
  increment: function() {
    this.counter++
    console.log(`> ${this.counter}`)
  }
}

const inc = obj.increment

inc()
inc()

console.log(obj.counter)
```

```javascript
const obj = {
  counter: 0,
  increment: function() {
    this.counter++
    console.log(`> ${this.counter}`)
  }
}

setInterval(obj.increment, 1000)
```

```
const obj = {
  counter: 0,
  increment: function() {
    this.counter++
    console.log(`> ${this.counter}`)
  }
}

const inc = obj.increment
setInterval(inc, 1000)
```

redradix

```javascript
global.name = 'Mr. Global'

const user = {
  name: 'Ms. Property',
  greet: function() {
    console.log(`Hi, I am ${this.name}`)
  }
}

user.greet()

const greet = user.greet
greet()
```

redradix

```
const counter = {
  count: 0,
  increment: function() { this.count++; }
}

$('#button').on('click', counter.increment)
```

```
const obj = {
  name: 'Homer',
  greet: function() {
    setTimeout(function() {
      console.log(`Hi, ${this.name}`)
    }, 100);
  }
}

obj.greet()
```

```
function greet() {
  console.log(`Hola, ${this.name}`)
}
const obj1 = {
  name: 'Homer'
}
const obj2 = {
  name: 'Fry'
}
```

Functions

There are four ways to invoke a function:

1. Direct invocation
2. Sending a message to an object (method)
3. **Function.prototype**

Functions

```
fn.call(context, arg1, arg2, …)

fn.apply(context, [arg1, arg2, …])
```

- Execute the function **fn**
- Specifying an **an explicit value for `this`**

```
function greet() {
  console.log(`Hi, ${this.name}`)
}
const obj1 = {
  name: 'Homer'
}

greet() // ???

obj1.greet() // ???

greet.call(obj1) // ???

setTimeout(greet.call(obj1), 1000) // ???
```

```
function add(a, b) {
  return a + b;
}

add(1, 1) // ???

add.call(1, 1) // ???

add.apply([], 1, 1) // ???

add.call(null, 1, 1) // ???

add.apply([null, 1, 1]) // ??
```

```
const obj = {
  name: 'Homer',
  greet: function() {
    console.log('Wait a second...')
    setTimeout(function() {
      console.log(`Hi, I am ${this.name}`)
    }, 1000)
  }
}

obj.greet()
```

```
function greet() {
  const self = this
  return function() {
    console.log(`Hi, I am ${self.name}`)
  }
}

const obj = { name: 'Homer' }

greet(obj) // ???
greet.call(obj) // ???
greet.call(obj)() // ???

const fn = greet.call(obj)
fn.call(null) // ???
fn.call({ name: 'Fry' }) // ??
```

What does this function do?

```
function mystery(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}
```

redradix

¿Qué hace esta función?

```
function mystery(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}

const something = mystery();

typeof something; // ???
typeof something(); // ???
```

redradix

¿Qué hace esta función?

```
function mystery(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}

const something = mystery({}, function() {
  return this;
});

typeof something(); // ???
```

¿Qué hace esta función?

```
function mystery(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}

const obj = {};
const something = mystery(obj, function() {
  return this;
});

obj === something(); // ???
```

¿Qué hace esta función?

```
function mystery(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}

const obj = {};
const something = mystery({}, function() {
  return this;
});

obj === something(); // ???
```

redradix

¿Qué hace esta función?

```
function mystery(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}

const obj = { name: 'Homer' };
const something = mystery(obj, function() {
  return this.name;
});

something(); // ???
```

redradix

¿Qué hace esta función?

```
function mystery(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}

const obj = { name: 'Homer' };
const something = mystery(obj, function(greet) {
  return `${greet}, ${this.name}`;
});

something('Hola'); // ???
```

¿Qué hace esta función?

```
function mystery(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}

const homer = { name: 'Homer' };
const fry = { name: 'Fry' };

const something = mystery(homer, function(greet) {
  return `${greet}, ${this.name}`;
});

something.call(fry, 'Hola'); // ???
```

redradix

```
function bind(ctx, fn) {
  return function() {
    return fn.apply(ctx, arguments);
  }
}
```

```
const obj = {
  name: 'Homer',
  greet: function() {
    setTimeout(bind(this, function() {
      console.log(`Hi, ${this.name}`)
    }), 100);
  }
}

obj.greet()
```

redradix

```
const obj = {
  name: 'Homer',
  greet: function() {
    setTimeout((function() {
      console.log(`Hola, ${this.name}`)
    }).bind(this), 100);
  }
}

obj.greet()
```

arrow functions

redradix

Arrow functions

- Alternative syntax to define anonymous functions
  - Shorter
  - More convenient
  - Safer

Arrow functions

```
(arg1, arg2, …) => { statement; statement; return …; }
```

Arrow functions

```
const add = (a, b) => {
  const result = a + b;
  return result;
};
```

Arrow functions

```
const add = (a, b) => {
  const result = a + b;
  return result;
};
```

redradix

Arrow functions

```
const add = (a, b) => {
  const result = a + b;
  return result;
};
```

redradix

Arrow functions

```
const add = (a, b) => {
  const result = a + b;
  return result;
};
```

redradix

Arrow functions

```
(arg1, arg2, …) => { statement; statement; return …; }

(arg1, arg2, …) => expression;
```

Arrow functions

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

redradix

Arrow functions

```
const add = (a, b) => { return a + b; };
```

redradix

Arrow functions

```
const add = (a, b) => ({ result: a + b });
```

redradix

Arrow functions

```
(arg1, arg2, …) => { statement; statement; return …; }

(arg1, arg2, …) => expression;

arg => expression;
```

Arrow functions

```
const random = n => Math.floor(Math.random() * n);
```

```
const obj = {
  name: 'Homer',
  greet: () => console.log(`Hi, I am ${this.name}`)
}

console.log(obj.greet()) // ???
```

```
const obj = {
  name: 'Homer',
  generateGreet: function(greet) {
    return () => {
      console.log(`${greet}, I am ${this.name}`)
    }
  }
}

const sp = obj.generateGreet(Hi)
sp() // ???
```

```
const greet = () => {
  console.log(`Hi, I am ${this.name}`)
}

const obj = { name: 'Homer' }

const binded = greet.bind(obj)

binded() // ???
```

redradix

```
const generator = {
  name: 'User Generator',
  createUser: function(name) {
    return { name, greet: () => console.log(`Hi, I am ${this.name}`) }
  }
}

const homer = generator.createUser('Homer')
```

redradix

Closures

```
let a = 1

function what() {
  return a
}
```

```
function what2() {
  {
    let a = 1
  }
  return a
}
```

redradix

```
function what3() {
  let a = 1
  return function() {
    return a
  };
}

let thing1 = what3()
```

```
function what3() {
  let a = 1
  return function() {
    return a
  };
}

let thing1 = what3()
console.log(thing1())
```

Closures

```
function counter() {
  return () => {
    let i = 0;
    return i++;
  };
}
```

redradix

Closures

```
const c1 = counter();
```

Closures

```
const c1 = counter();
console.log(c1());
```

Closures

```
const c1 = counter();
console.log(c1()); // 0
```

Closures

```
const c1 = counter();
console.log(c1()); // 0
console.log(c1());
```

redradix

Closures

```
const c1 = counter();
console.log(c1()); // 0
console.log(c1()); // 0
console.log(c1()); // 0
```

Closures

```
const c1 = counter();
```

c1

```
() => {
  let i = 0;
  return i++;
};
```

Closures

```
const c1 = counter();
c1();
```

c1

```
() => {
  let i = 0;
  return i++;
};
```

redradix

Closures

```
const c1 = counter();
c1();
```

c1

```
() => {
  let i = 0;
  return i++;
};
```

```
i = 0
```

Closures

```
const c1 = counter();
c1(); // 0
```

c1

```
() => {
  let i = 0;
  return i++;
};
```

```
i = 1
```

Closures

```
const c1 = counter();
c1(); // 0
```

c1

```
() => {
  let i = 0;
  return i++;
};
```

~~i = 1~~

Closures

```
const c1 = counter();
c1(); // 0
c1();
```

c1

```
() => {
  let i = 0;
  return i++;
};
```

```
i = 0
```

Closures

```
const c1 = counter();
c1(); // 0
c1(); // 0
```

c1

```
() => {
  let i = 0;
  return i++;
};
```

```
i = 0
```

Closures

```
function counter() {
  let i = 0;
  return () => i++;
}
```

redradix

Closures

```
function counter() {
  let i = 0;
  return () => i++;
}
```

Closures

```
function counter() {
  let i = 0;
  return () => i++;
}
```

redradix

Closures

```
function counter() {
  let i = 0;
  return () => i++;
}
```

redradix

Closures

```
function counter() {
  let i = 0;
  return () => i++;
}
```

Closures

```
function counter() {
  let i = 0;
  return () => {
    i++;
    return i;
  };
}
```

redradix

Closures

```
const c1 = counter();
console.log(c1());
```

Closures

```
const c1 = counter();
console.log(c1()); // 0
console.log(c1());
```

Closures

```
const c1 = counter();
console.log(c1()); // 0
console.log(c1()); // 1
console.log(c1()); // 2
```

Closures

```
const c1 = counter();
```

c1

```
() => i++;
```

Closures

```
function counter() {
  let i = 0;
  return () => i++;
}
```

redradix

Closures

```
const c1 = counter();
let i = 10;
c1();
```

c1

```
() => i++;
```

redradix

Closures

```
const c1 = counter();
let i = 10;
c1();                // ???
console.log(i); // ???
```

c1

```
() => i++;
```

redradix

Closures

```
const c1 = counter();
let i = 10;
c1();                // 0
console.log(i); // 10
```

c1

```
() => i++;
```

```
i = ??
```

Closures

```
function counter() {
  let i = 0;
  return () => i++;
}
```

Closures

```
const c1 = counter();
c1(); // 0
c1(); // 1
```

c1

```
() => i++;
```

```
i = 2
```

Closures

```
const c1 = counter();
c1(); // 0
c1(); // 1

const c2 = counter();
c2(); // ???
```

c1

```
() => i++;
```

```
i = 2
```

c2

```
() => i++;
```

```
i = 1
```

redradix

Closures

- Variables, in javascript, have *indefinite extent*
  - Persist for as long they are **reachable**
  - Only destroyed when it **becomes impossible to access** them
- A free variable keeps a reference to the original variable and keeps it alive
- As long as the free variable is still reachable (directly or indirectly)
- This is phenomenon is what we call **closure**

Closures

```
function counter() {
  let i = 0;
  return () => i++;
}
```

redradix

Closures

```
const c1 = counter();
c1(); // 0
c1(); // 1

const c2 = counter();
c2(); // 0
```

c1

`() => i++;`	`i = 1`

c2

`() => i++;`	`i = 0`

Constructors

Constructors

There are four ways to invoke a function:

1. Direct invocation
2. Sending a message to an object (method)
3. Function.prototype
4. **new**

redradix

Constructors

- A function is called as a constructor when the invocation is preceded by the word **new**
- **Before** running the constructor function, **three things** happen:

Constructors

1. A new, **empty** object is created
2. The prototype of the new object is the ***value of the "prototype" property*** *of the constructor*
3. **`this`** inside the constructor points to the new object

Constructors

- And the, the constructor funciton is executed
- The value of the expression **`new Constructor()`** is:
  - The **return** value of the function
  - If the constructor **does not return anything**, the the **new object** is returned implicitly

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.bark = function() {
  console.log("wof, wof...");
}

Dog.prototype.sit = function() {
  console.log(`* ${this.name} sits and looks at you.`);
}

const toby = new Dog("Toby");
toby.sit();
```

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.bark = function() {
  console.log("wof, wof...");
}

Dog.prototype.sit = function() {
  console.log(`* ${this.name} sits and looks at you.`);
}

const toby = new Dog("Toby");
toby.sit();
```

```javascript
function Dog(name) {
  this.name = name;
}

Dog.prototype.bark = function() {
  console.log("wof, wof...");
}

Dog.prototype.sit = function() {
  console.log(`* ${this.name} sits and looks at you.`);
}

const toby = new Dog("Toby");
toby.sit();
```

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.bark = function() {
  console.log("wof, wof...");
}

Dog.prototype.sit = function() {
  console.log(`* ${this.name} sits and looks at you.`);
}

const toby = new Dog("Toby");
toby.sit();
```

Constructors

true or false?

```
toby.hasOwnProperty("name")
```

Constructors

true or false?

```
toby.hasOwnProperty("sit")
```

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.bark = function() {
  console.log("wof, wof...");
}

Dog.prototype.sit = function() {
  console.log(`* ${this.name} sits and looks at you.`);
}

const toby = new Dog("Toby");
toby.sit();
```

Constructors

- Each instance holds its own state
- They share the method implementations through their prototype

redradix

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.sit = function() {
  console.log(`* ${this.name} sits and looks at you.`);
}

const toby = new Dog("Toby");

Dog.prototype.sit = function() {
  console.log(`* ${this.name} does not understand.`);
}

const spot = new Dog("Spot");
spot.sit();
```

redradix

```javascript
function Dog(name) {
  this.name = name;
}

Dog.prototype.sit = function() {
  console.log(`* ${this.name} sits and looks at you.`);
}

const toby = new Dog("Toby");

Dog.prototype.sit = function() {
  console.log(`* ${this.name} does not understand.`);
}

const spot = new Dog("Spot");
spot.sit();
toby.sit();
```

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.sit = () => {
  console.log(`* ${this.name} sits and looks at you.`);
}

const toby = new Dog("Toby");
toby.sit();
```

redradix

Exercise

- Write a constructor **User**, that receives a **name** as a parameter and has a method **greet** that shows a greeting with its name

Exercise

- Write a constructor **Root** in such a way that only **one instance** can be created

```
function User(name) {
  this.name = name
  this.usersCreated++
}

User.prototype = {
  greet: function() {
    console.log(`Hi, I am ${this.name}`)
  },
  getTotalUsers: function() {
    return this.usersCreated
  },
  usersCreated: 0
}
```

```js
function User(name) {
  this.name = name
}

User.prototype = {
  greet: function() {
    console.log(`Hi, I am ${this.name}`)
  }
}

const homer = new User('Homer')
const fry = new User('Fry')
```

redradix

```
const homer2 = new User('Homer')
console.log(homer === homer2) // ???

console.log(homer.greet === homer2.greet) // ???

homer2.greet = () => console.log('Good morning')

homer.greet() // ??? (why??)

User.prototype.greet = () => console.log('Hola!')
fry.greet() // ???
homer2.greet() // ???
```

redradix

Exercise

- Write the function **myNew(Constructor, ...params)**
  - replicates the behaviour of **new**
  - using **Object.create**

Exercise

- Write the function **withCount**
    - *(see next slide)*

```
function User(name) {
  this.name = name
}

User.prototype = {
  greet: function() {
    console.log(`Hi, I am ${this.name}`)
  }
}

const CountedUser = withCount(User);
const u1 = new CountedUser('Homer')
const u2 = new CountedUser('Fry')

u1.greet() // 'Hi, I am Homer'

CountedUser.getInstanceCount() // 2
```

```
function Animal(species, color) {
  this.species = species
  this.color = color
}

Animal.prototype = {
  toString: function() {
    return `A ${this.color} ${this.species}`
  },
  getSpecies() {
    return this.species
  }
}
```

```
function Dog(color, name) {
  this.name = name
  // ???
}

Dog.prototype = {
  toString: function() {
    // ???
  }
}

var toby = new Dog('green', 'Toby');
toby.getSpecies() // 'dog'
toby.toString() // 'A green dog called Toby'
```

```
console.log(toby instanceof Perro) // ???
console.log(toby instanceof Animal) // ???
console.log(toby instanceof Object) // ???

console.log(Perro instanceof Animal) // ???
console.log(Perro instanceof Function) // ???
```

Exercise

- Starting from the **`Container`** class...
  - Write two derived constructors:
    - **`ItemContainer`**
    - **`NestedContainer`**

redradix

```
function Container(name) {
  this.name = name
}

Container.prototype = {
  canFit: function(item) {
    throw new Error('Abstract method')
  },
  store: function(item) {
    throw new Error('Abstract method')
  },
  retrieve: function(index) {
    throw new Error('Abstract method')
  }
}
```

Exercise

- `ItemContainer(`*`name`*`)`
  - Inherits from **`Container`**
  - Contains **`Items`**
  - Implement the abstract methods of **`Container`**
  - Can hold an infinite number of **items**

```
function Item(name, size, category, createdAt) {
  Object.assign(this, { name, size, category, createdAt })
}

Item.prototype.getSize = function() { return this.size }
```

```
const itemContainer = new ItemContainer('Test Container')

const item1 = new Item('Item1', 10, 'test', new Date())

itemContainer.canFit(item1) // true
const index = itemContainer.store(item1)
console.log(index) // [0]

const retrieved = itemContainer.retrieve(index)
console.log(retrieved.name) // Item1
```

redradix

Exercise

- **`ItemBox(`*`capacity`*`)`**
  - Inherits from **`ItemContainer`**
  - Has limited capacity
    - Given as parameter to the constructor
  - Each stored **`item`** uses some space
    - Item property **.size**
  - The sum of all items **`items`** stored can’t exceed the capacity

```
const box = new ItemBox(10)

const item1 = new Item('Item 1', 5, 'test', new Date())
const item2 = new Item('Item 2', 3, 'test', new Date())
const item3 = new Item('Item 3', 3, 'test', new Date())

box.store(item1)
box.store(item2)

box.canFit(item3) // false

console.log(box.retrieve([1]).name) // Item 2
```

Exercise

- **`NestedContainer(`*`name`*`, `*`subcontainers`*`)`**
  - Inherits from **`Container`**
  - Contains **`Containers`**
  - Implements the abstract methods of **`Container`**
  - Receives the sub-containers in the constructor

Exercise

- `NestedContainer(`*`name`*`, `*`subcontainers`*`)`
  - `store(item)`
    - Delegates in the first sub-container that can hold **item**
  - `canFit(item)`
    - Does **item** fit in any of the sub-container?
  - `retrieve(index)`
    - **index** is an **array** with multiple numbers
    - The first element of the array is the sub-container index

```
const boxes = [new ItemBox(10), new ItemBox(10)]
const nestedContainer = new NestedContainer('NestedContainer', boxes)

const item1 = new Item('Item 1', 5, 'test', new Date())
const item2 = new Item('Item 2', 3, 'test', new Date())
const item3 = new Item('Item 3', 3, 'test', new Date())
const item4 = new Item('Item 4', 8, 'test', new Date())
```

```
nestedContainer.store(item1)
const i1 = nestedContainer.store(item2)
console.log(i1) // [0, 1]

nestedContainer.canFit(item3) // true
const i2 = nestedContainer.store(item3)

console.log(i2) // [1, 0]

nestedContainer.canFit(item4) // false

console.log(nestedContainer.retrieve([0, 1]).name) // Item 2
```

Exercise

- Derived from `NestedContainer`…
  - `Shelf`
    - Set of `ItemBoxes`
  - `Rack`
    - Set of `Shelf`
  - `Warehouse`
    - Set of `Rack`

Exercise

- `Shelf(`***`maxBoxes`***`, `***`boxCapacity`***`)`
    - Starts *empty* (zero boxes)
    - Boxes are created on-demand
    - No more than **maxBoxes** boxes

```
const shelf = new Shelf(2, 10)

const item1 = new Item('Item 1', 5, 'test', new Date())
const item2 = new Item('Item 2', 3, 'test', new Date())
const item3 = new Item('Item 3', 3, 'test', new Date())
const item4 = new Item('Item 4', 8, 'test', new Date())
const item5 = new Item('Item 5', 1, 'test', new Date())

// shelf starts with 0 boxes...
console.log(shelf.subcontainers.length) // 0

// ...but has to create a new box to hold item1
shelf.canFit(item1) // true
shelf.store(item1)
console.log(shelf.subcontainers.length) // 1
```

```
shelf.canFit(item2) // true
shelf.store(item2)
console.log(shelf.subcontainers.length) // 1

shelf.canFit(item3) // true
shelf.store(item3)
console.log(shelf.subcontainers.length) // 2

shelf.canFit(item4) // false

shelf.canFit(item5) // true
console.log(shelf.store(item5)) // [0, 2]
```

Exercise

- `Rack(`*`numShelves`*`, `*`boxesPerShelf`*`, `*`boxCapacity`*`)`
    - Starts with **numShelves** empty instances of **Shelf**
        - the instances are created in the constructor

```
const rack = new Rack(2, 2, 5)

const item1 = new Item('Item 1', 5, 'test', new Date())
const item2 = new Item('Item 2', 3, 'test', new Date())
const item3 = new Item('Item 3', 3, 'test', new Date())
const item4 = new Item('Item 4', 8, 'test', new Date())
const item5 = new Item('Item 5', 1, 'test', new Date())

console.log(rack.subcontainers.length) // 2

rack.store(item1)
rack.store(item2)
console.log(rack.store(item3)) // [1, 0, 0]

rack.canFit(item4) // false
rack.canFit(item5) // true

console.log(rack.retrieve([0, 1, 0]).name) // Item 2
```

Exercise

- `Warehouse(racks)`
  - Receives a *configuration of Racks*
  - Its peculiarities:
    - In the **.store(...)** method
      - raises an exception if trying to store an **item** that doesn't fit in any of the sub-containers

```
const warehouse = new Warehouse([
  new Rack(2, 2, 5),
  new Rack(2, 1, 10)
])

const item1 = new Item('Item 1', 5, 'test', new Date())
const item2 = new Item('Item 2', 3, 'test', new Date())
const item3 = new Item('Item 3', 3, 'test', new Date())
const item4 = new Item('Item 4', 8, 'test', new Date())
const item5 = new Item('Item 5', 1, 'test', new Date())

console.log(warehouse.store(item1)) // [0, 0, 0, 0]
warehouse.store(item2)
warehouse.store(item3)

warehouse.canFit(item4) // true
console.log(warehouse.store(item4)) // ????

console.log(warehouse.retrieve([0, 0, 1, 0]).name) // Item 2
```

redradix

```
const warehouse = new Warehouse([
  new Rack(2, 2, 5),
  new Rack(2, 1, 10)
])

const item1 = new Item('Item 1', 5, 'test', new Date())
const item2 = new Item('Item 2', 3, 'test', new Date())
const item3 = new Item('Item 3', 3, 'test', new Date())
const item4 = new Item('Item 4', 8, 'test', new Date())
const item5 = new Item('Item 5', 1, 'test', new Date())

console.log(warehouse.store(item1)) // [0, 0, 0, 0]
warehouse.store(item2)
warehouse.store(item3)

warehouse.canFit(item4) // true
console.log(warehouse.store(item4)) // ????

console.log(warehouse.retrieve([0, 0, 1, 0]).name) // Item 2
```

redradix

Classes

```
class User {
  constructor(name) {
    this.name = name
  }

  greet() {
    console.log(`Hi, I am ${this.name}`)
  }
}
```

```
class Root extends User {
  constructor() {
    // MANDATORY to call super from the constructor
    super('ROOT')
  }
  greet() {
    super.greet()
  }
}
```

redradix

```
class Root extends User {
  constructor() {
    // MANDATORY to call super from the constructor
    super('ROOT')
  }
  greet() {
    super.greet()
  }
}
```

Exercise

- Rewrite the **Animal** and **Dog** example using **class** and **extend**

Exercise

- Translate all the constructors to classes:
  - **Warehouse**, **Rack**, **Shelf**, **ItemBox**, **ItemContainer**, **NestedContainer**, **Item** and **Container**

```javascript
class User {
  constructor(name) {
    this.name = name
  }
  greet() {
    console.log(`Hi, I am ${this.name}`)
  }
}

const u1 = new User('Homer')
const u2 = new User('Fry')
```

```
u1.greet.call(u2) // ???

u2.greet = u1.greet

u2.greet() // ???

User.prototype.greet = () => console.log('How do you do?')

u1.greet() // ???

u2.greet() // ???
```

Anonymous Classes

- **class** can be used as an expression
- To create dynamic or anonymous classes

Anonymous Classes

```
const Mammal = class {
  constructor(name) {
    this.name = name;
  }
}

const buddy = new Mammal('Buddy');
console.log(buddy.name)
```

Exercise

- Rewrite **withCount** for classes

Iterators

Iterators

- Interface (`iterable protocol`)
- How to go through the elements of a collection
  - any collection, not just `Array`
- Tight integration with the language
  - `for...of`
  - `Array.from(...)`

Iterators

- Any object
- With a method `.next()`
- That returns *an object* with two properties:
  - `value`
  - `done`

Iterators

```
let i = 0;

const iterator = {
  next: () => {
    return { done: false, value: i++ };
  }
}
```

Iterators

```
function makeIterator(array) {
  let i = 0;
  return {
    next: () => {
      const done = i === array.length;
      return { done, value: (done || array[i++]) };
    }
  };
}
```

redradix

Iterators

```
function makeIterator(array) {
  let i = 0;
  return {
    next: () => {
      const done = i === array.length;
      return { done, value: (done || array[i++]) };
    }
  };
}
```

redradix

Iterators

```
function makeIterator(array) {
  let i = 0;
  return {
    next: () => {
      const done = i === array.length;
      return { done, value: (done || array[i++]) };
    }
  };
}
```

redradix

Iterators

```
let i = makeIterator([1, 2, 3, 4]);

console.log(i.next()); // { value: 1, done: false }
console.log(i.next()); // { value: 2, done: false }
console.log(i.next()); // { value: 3, done: false }
console.log(i.next()); // { value: 4, done: false }
console.log(i.next()); // { value: true, done: true }
```

redradix

Iterators

```
let i = makeIterator([1, 2, 3, 4]);

let next = i.next();
while (!next.done) {
  console.log(next.value);
  next = i.next();
}
```

redradix

Iterators

```
let i = makeIterator([1, 2, 3, 4]);

for (let n = i.next(); !n.done; n = i.next()) {
  console.log(n.value);
}
```

redradix

Iterators: Exercise

- Implement an iterator that…
  - …returns the elements of an array in random order
  - …returns the numbers of a given range (from, to)
  - …returns the (infinite!) fibonacci series

Iterators

- An **iterable** is a data structure...
  - With a function in `[Symbol.iterator]`
  - That returns an **iterator**
  - That goes through every element in the structure

Iterators

- Javascript knows how to go over **iterables**
  - `for … of`
  - `Array.from(…)`
- Many native objects are **iterable**
  - `Array`
  - `Map`
  - `...`

Iterators

```
const list = [1, 2, 3, 4];

for (const item of list) {
  console.log(item);
}
```

redradix

Iterators

- To make our own **iterables:**
  - Store inside `[Symbol.iterator]`...
  - a function
  - that returns an **iterator**

Exercise

- Make **ItemContainer** ***iterable***
    - returns every one of its stored **items**

```
const box = new ItemBox(10)

box.store(new Item('Item 1', 3, 'test', new Date()))
box.store(new Item('Item 2', 3, 'test', new Date()))
box.store(new Item('Item 3', 1, 'test', new Date()))

for (const item of box)
  console.log(item.name) // logs every item name
```

Exercise

- Make **NestedContainer** ***iterable***
  - returns every **item** stored…
  - … in **all of its sub-containers**!

```javascript
const warehouse = new Warehouse([
  new Rack(2, 2, 5),
  new Rack(2, 1, 10)
])

const item1 = new Item('Item 1', 5, 'test', new Date())
const item2 = new Item('Item 2', 3, 'test', new Date())
const item3 = new Item('Item 3', 3, 'test', new Date())
const item4 = new Item('Item 4', 8, 'test', new Date())
const item5 = new Item('Item 5', 1, 'test', new Date())

warehouse.store(item1)
warehouse.store(item2)
warehouse.store(item3)
warehouse.store(item4)
warehouse.store(item5)

for (const item of warehouse)
  console.log(item.name) // logs every item name
```

Iterators

```
let i = {
  [Symbol.iterator]: () => makeIterator([1, 2, 3, 4])
};
```

Iterators

```
for (const v of i) {
  console.log(v);
}
```

Generators

Generators

- A *special function*
- Behaves as an **iterator factory**
  - Returns an iterator when executed
  - Greatly simplifies the implementation of iterators
  - Because of how it handles the iteration state

Generators

- Dedicated syntax:
  - `function*`
  - `yield`

Generators

```
function* generator() {
  yield 1
  yield 2
}
```

redradix

Generators

```
const i = generator()
```

Generators

```
const i = generator()

let n = i.next()
console.log(n.value) // 1
```

Generators

```
const i = generator()

let n = i.next()
console.log(n.value) // 1

n = i.next()
console.log(n.value) // 2
```

redradix

Generators

```
const i = generator()

let n = i.next()
console.log(n.value) // 1

n = i.next()
console.log(n.value) // 2

n = i.next()
console.log(n.value) // undefined
console.log(n.done) // true
```

redradix

Generators

```
function* generator() {
  yield 1
  return 2
}
```

redradix

Generators

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

Generators

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

redradix

Generators

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

Generators

```
for (let n of range(10, 20))
  console.log(n);
```

Generators

```
function* peculiar() {
  console.log('Give me a 1!');
  yield 1;
  console.log('Give me a 2!');
  yield 2;
  console.log('Give me a 3!');
  yield 3;
}
```

redradix

Generators

```
const i = peculiar();
```

Generators

```
const i = peculiar();

let n = i.next(); // Give me a 1!
console.log(n.value); // 1
```

Generators

```
const i = peculiar();

let n = i.next(); // Give me a 1!
console.log(n.value); // 1

n = i.next(); // Give me a 2!
console.log(n.value); // 2
```

redradix

Exercise

- Make **NestedContainer** ***iterable***
    - Using a **generator**
    - Tip:
        - The execution of a *generator* returns an ***iterator***

Sets

Sets

`new Set(iterable)`

- Stores unique values
  - Primitive
  - By reference (`object`)
- Not a native type
  - **`typeof`** tags them as `'object'`

Sets

```
const s = new Set();
s.add('A');
console.log(s.has('A')); // true
console.log(s.has('B')); // false
```

Sets

- `add(value)`
- `delete(value)`
- `clear()`
- `has(value)`

Sets

```
const s2 = new Set(['A', 'B']);
console.log(Array.from(s2));
```

Sets

```
const s2 = new Set(['A', 'B']);

for (let value of s2) {
  console.log(value);
}
```

Sets

```
const s2 = new Set(['A', 'B']);

for (let value of s2) {
  console.log(value);
}
```

Exercise

- Implement the three basic set operations:
  - `union(A, B)`
  - `intersection(A, B)`
  - `difference(A, B)`

Ejercicio

```
> const t1 = new Set(['A', 'B'])

> const t2 = new Set(['C', 'B'])

> union(t1, t2) // Set { 'A', 'B', 'C' }

> intersection(t1, t2) // Set { 'B' }

> difference(t1, t2) // Set { 'A' }

> difference(t2, t1) // Set { 'C' }
```

Maps

Maps

`new Map(`*`iterable`*`)`

- Stores key-value pairs
- Dictionaries
- Not a native type
  - **`typeof`** tags them as `'object'`

Maps

```
const m = new Map();
m.set('clave', 'valor');
console.log(m.get('clave'));
```

Maps

```
const m = new Map();
m.set('clave', 'valor');
console.log(m.get('clave'));
```

redradix

Maps

```
const m = new Map([['a', 1], ['b', 2]]);
```

Maps

```
const m = new Map([['a', 1], ['b', 2]]);
console.log(Array.from(m));
```

redradix

Maps

- `.set(`*`key, value`*`)`
- `.get(`*`key`*`)`
- `.has(`*`key`*`)`
- `.delete(`*`key`*`)`
- `.clear()`
- `.size`

redradix

Maps

```
const m = new Map([['a', 1], ['b', 2]]);

console.log(m.has('a')); // true
console.log(m.has('c')); // false

m.delete('b'); // true
m.delete('c'); // false

console.log(m.get('b')); // undefined
console.log(m.size); // 1
```

redradix

Maps

To go over a map...

- `.keys()`
- `.values()`
- `.forEach(`*`fn`*`)`
- `.entries()`
- The instance is iterable

Maps

```
const m = new Map([['a', 1], ['b', 2]]);

console.log(Array.from(m.keys())); // [ 'a', 'b' ]
console.log(Array.from(m.values())); // [ 1, 2 ]
```

redradix

Maps

```
const m = new Map([['a', 1], ['b', 2]]);

m.forEach(function(valor, clave) {
  console.log(clave + ' -> ' + valor);
});

// a -> 1
// b -> 2
```

redradix

Maps

`Map` > `Object`

- Better semantics
  - Cleaner API
  - More explicit intent

Maps

`Map` > `Object`

- Not affected by prototype inheritance
    - Only shows own keys

Maps

`Map` > `Object`

- Keep the insertion order
    - Objects are not guaranteed to keep the insertion order

redradix

Maps

`Map` > `Object`

- Richer and more convenient API
  - `.size`
  - `.has(...)`
  - `.clear(...)`
  - `...`

redradix

Maps

`Map` > `Object`

- Start *empty*
    - “empty” objects have a lot of inherited properties
        - `.constructor`, `.toString`, ….

Maps

`Map` > `Object`

- **Any value** can be used as key
  - Not limited to `String` or `Symbol`

Maps

```
const m = new Map();
m.set({ a: 1 }, 'value');
```

Maps

```
const m = new Map();
m.set({ a: 1 }, 'value');

console.log(m.get({ a: 1 })); // ???
```

redradix

Maps

```
const m = new Map();
const k = { a: 1 };
m.set(k, 'value');
```

Maps

```
const m = new Map();
const k = { a: 1 };
m.set(k, 'value');

console.log(m.get(k)); // ???
```

Maps

```
const a = new Map([['a', 1], ['b', 2]]);
const b = new Map([['a', 1], ['b', 2]]);

console.log(a === b); // ???
```

Exercise

- Implement these operations:
  - `merge(A, B, C, ...)`
  - `equal(A, B)`
  - `deepEqual(A, B)`

Destructuring

Destructuring

- Special **syntax** that allow us to **“disassemble”** a given data structure
- To be able to reference its inner values
- Describing the “position” they hold inside the structure

```
const [a, b] = [1, 2]
```

```
const { x, y } = { x: 10, y: 20 }
```

Exercise

- Disassemble the object `{ one: 1, two: 2 }` into two variables: `one` and `two`

Exercise

- Use destructuring to **swap the value** of the variables **a** and **b** *without creating any other variable*

```
let a = 1
let b = 2
// ???

console.log(a, b) // “2 1”
```

```
const { x: equis, y: igriega } = { x: 10, y: 20 }
```

```
const { x: { y } } = { x: { y: 10 } }
```

Exercise

- Disassemble this object into the variables **one**, **two**, **three**, **four** and **five**

```
{ one: 1, list: [2, 3], four: 4, x: { five: 5 } }
```

Exercise

- Disassemble this object into the variables **a**, **b**, **c**, **d** and **e**

```
{ one: 1, list: [2, 3], four: 4, x: { five: 5 } }
```

```
const [head] = [1, 2, 3]
```

```
const [, , tres] = [1, 2, 3]
```

Exercise

- Make a **data structure** that can be disassembled with the following expression:

```
const [{ list: [ , { x: { y: two } } ] }] = structure
```

```
const [head, ...tail] = [1, 2, 3]
```

```
const [head, tail] = [ 1, 2, 3]

const [head, ...tail] = [1, 2]

const [head, ...tail] = [1]

const [head, , ...tail] = [1, 2, 3]
```

```
const lista1 = [1, 2, 3]
const [...lista2] = lista1

lista1 === lista2 // ??

let [a, b, c] = lista1
let [x, y, z] = lista2
a === x && b === y && c === z // ???
[a, b, c] === lista1 // ???

[a, b, c] = [x, y, z]
a === x && b === y && c === z // ???

[c, b, a] = [a, b, c]
a === x && b === y && c === z // ???
```

```
const list = [1, 2, 3];

console.log(list) // [1, 2, 3]

console.log(...list) // 1 2 3

console.log(1, 2, 3) // 1 2 3
```

```
const list1 = [1, 2]
const list2 = [3, 4]
const a = [list1, list2] // ???
const b = [...list1, ...list2] // ???
```

```
const [a, b, c = 3] = [1, 2]
console.log(a, b, c) // 1 2 3
```

redradix

```
const { x: { y = 1 } = { y: 2 } } = { x: { y: 3 } }
const { x: { y = 1 } = { y: 2 } } = { x: { z: 3 } }
const { x: { y = 1 } = { y: 2 } } = { }

const [y = 10] = [2]
const [y = 10] = []
const [y = 10] = [1, 2]
const [y = 10] = [false]
const [y = 10] = [null]
const [y = 10] = [undefined]
```

redradix

```
function add(a = 1, b = 1) {
  return a + b;
}

add() // 2
add(2) // 3
add(2, 2) // 4
```

```
function someFunc({ x: equis, y: igriega = 10 }) {
  return equis + igriega;
}

someFunc({ x: 1, y: 10 }) // 11
someFunc({ x: 1 }) // 11
```

```
function addAll(...args) {
  let total = 0;
  while (args.length) total += args.pop();
  return total;
}

addAll(1)
addAll(1, 1, 1, 1)
```