SUPINFO International University

Sign of success

Advanced JavaScript

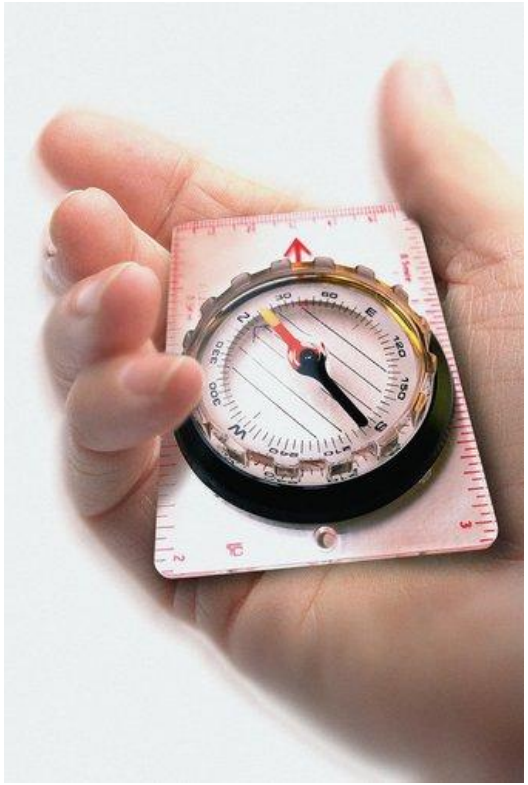
Go further with JavaScript…

Course objectives

By completing this course, you will be able to:

- Explain prototype-based OOP
- Develop with JavaScript OOP
- Manipulate Property Descriptors
- Explain what is "strict mode" and use it
- Use the most famous JS pattern

Course topics



Course's plan:

- Reminders
- Object Oriented Programming
- Inheritance
- Property Descriptors
- Strict Mode
- Good Practices

Go further with JavaScript…

REMINDERS

Presentation

- JavaScript is a scripting language

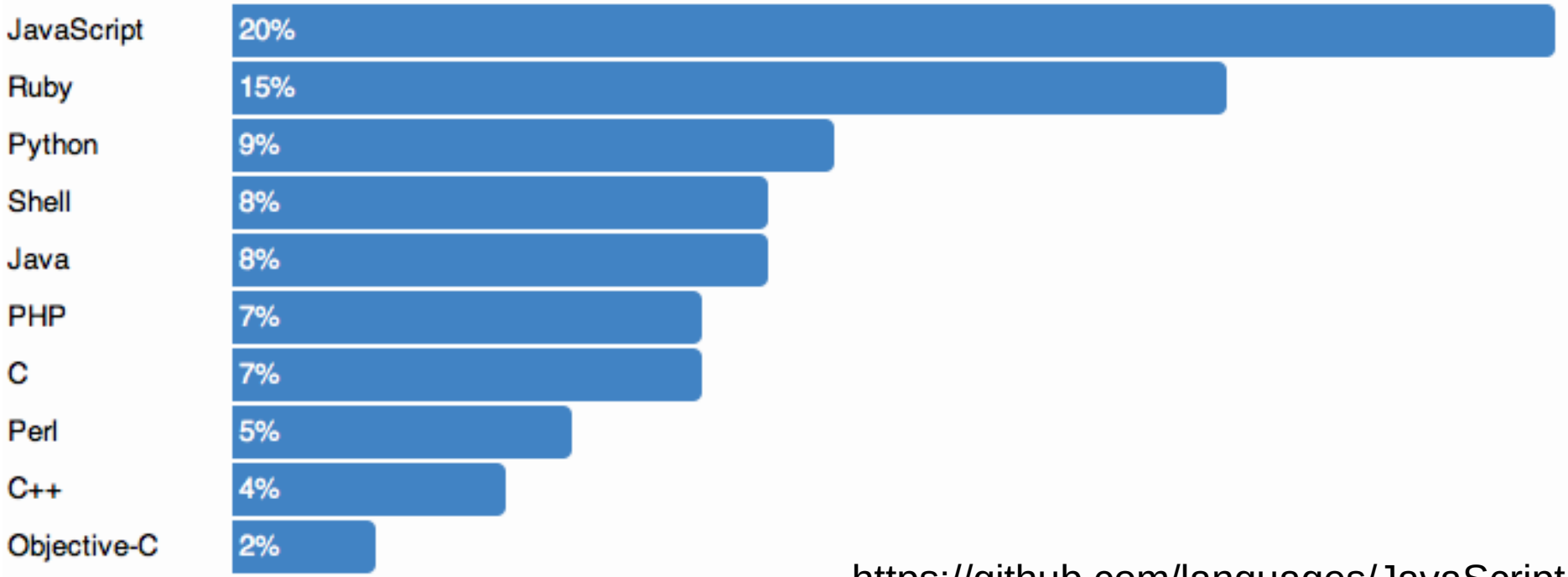
- Mostly known for building browser-based applications
  - User Interactions
  - Animations
  - …

Reminders

Community

Reminders

Community

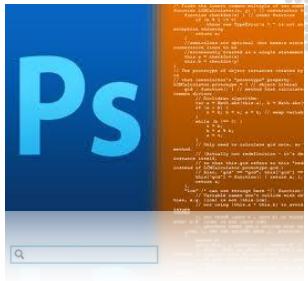


https://github.com/languages/JavaScript

Reminders

JavaScript Everywhere













Syntax & Types

- JavaScript uses syntax influenced by that of C

- A weakly typed programming language
  - No need to specify the type
  - Support implicit type conversion

Syntax & Types

- Example:

```
function computeAverage(values) {
        var i, sum = 0, length = values.length;
        for(i = 0; i < length; i += 1) {
                sum += values[i];
        }
        return sum / length;
}

var marks = [12, 18, 14, 8];
console.log(computeAverage(marks)); // 13
```

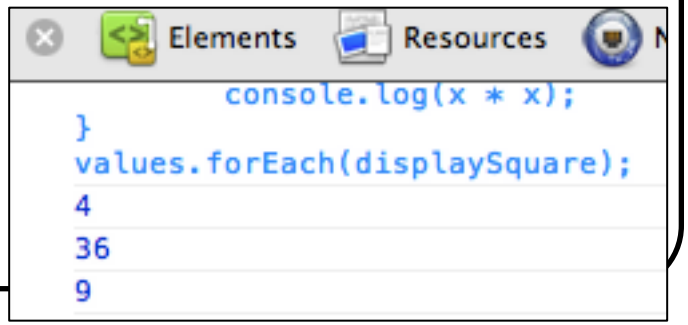
Variable scope

- Local:
  - Reachable only in the function where it's defined
- Global:
  - Reachable in the whole document
- function = scope
  - And not *block = scope* like in Java or C#

Function Expressions

- JavaScript supports also function expressions
  - Functions with or without name (anonymous)
  - Can be used to contain functionality for short-term use

```
var values = [2, 6, 3];
var displaySquare = function(x) {
   console.log(x * x);
}
values.forEach(displaySquare);
```



Functional

- JavaScript is a functional language!

- First-class functions:
  - Can be assigned to variables or stored in data structures
  - Can be passed as arguments to other functions
  - Can be returned as the values from other functions

Reminders

Functional

- Example:

```
var computeAverage = function(values) {
        var sum = values.reduce( function(total, current) {
                return total + current;
        }, 0);
        return sum / values.length;
}

var marks = [12, 18, 14, 8];
console.log(computeAverage(marks)); // 13
```

Fn expression VS Fn declaration

- Function declarations are evaluated before any instructions in the same context

- Function expressions are evaluated after all the instructions preceding it

Questions ?

Go further with JavaScript…

OBJECT ORIENTED PROGRAMMING

Presentation

- JavaScript is an Object Oriented Programming language that uses Prototypes
  - OOP style with prototype instead of classes

Presentation

- Do you remember the analogy between a class and a plan?
  - A class is an object type plan
  - We use that plan to create new instances

- In prototype-based OOP, we don't use plan…
  - We create objects from scratch…
  - … or based on other objects (prototype)

Presentation

- Objects in JavaScript are mutable keyed collections
- *Numbers*, *strings*, *booleans*, *null* and *undefined* are simple types
- *Arrays*, *RegEx* and even *Functions* are objects

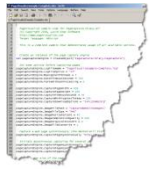
Presentation

- In JavaScript, an object is a set of properties
  - Can be methods or instance variables

- A property has a name and a value

Presentation

- A property name can be any string
- A property value can be any JavaScript value
  - Strings, Numbers, Functions, …
  - *undefined* by default
- JavaScript provide several ways to declare objects…



Object Literals

- Convenient notation for creating new objects

- A pair of curly braces surrounding zero or more name/value pairs:

```
var barney = {
        "firstName": "Barney",
        "lastName": "Stinson",
        "saySmthg": function() {
                console.log("It's gonna be...");
        }
}
```

Object Literals

- Quotes around property names are optional if the name is a legal JavaScript identifier

- Property values can be other object literals

Object Literals

- Example:

```
var trip = {
        departure: {
                city: "Paris",
                country: "France"
        },
        arrival: {
                city: "Montreal",
                country: "Canada"
        },
        price: 890
}
```

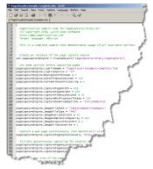
Object Literals

- To access a property:

```
var firstName = barney.firstName;

var lastName = barney["lastName"];
```

- To call a method:

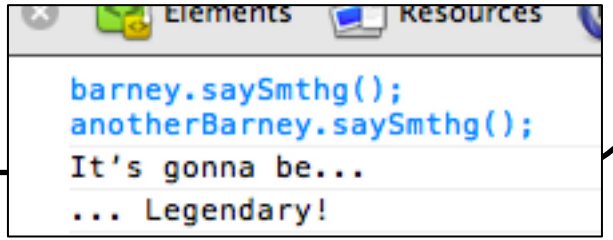
```
barney.saySmthg();
```

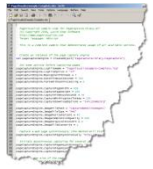


Object Literals

- To create new instances based on an existing one, you can clone it:

```
var anotherBarney = Object.create(barney);

anotherBarney.saySmthg = function() {
        console.log("... Legendary!");
};

barney.saySmthg();
anotherBarney.saySmthg();
```

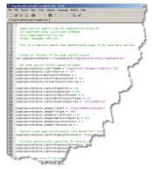




Prototype link

- Every object is linked to a prototype from which it can inherit properties
  - Object literals are linked to *Object.prototype*
- Similar to inheritance…

```
Object.getPrototypeOf(barney) === Object.prototype; // true

// toString is a inherited method from Object.prototype
barney.toString === Object.prototype.toString; // true
```

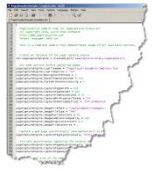


Prototype link

- When an object is used with *Object.create()* to create a new instance
  - The original object become the prototype of the new one

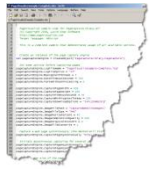
```
Object.getPrototypeOf(anotherBarney) === barney; // true
```

- That’s why the new object can access to the properties of the original!



Function Objects

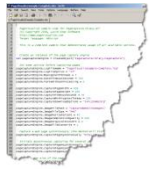
- Functions in JavaScript are objects
- You can also use them to define objects
  - An alternative to Object Literals!



Function Objects

- Example :

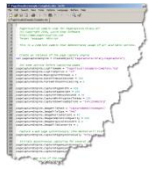
```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
};
Person.prototype.sayHello = function() {
    console.log("Hey! My name is " + this.firstName + ".");
}

var eric = new Person("Eric", "Cartman");
var johnDoe = new Person("John", "Doe");

johnDoe.sayHello();
```



Function Objects

- Functions that are intended to be used with the *new* prefix are called *constructors*

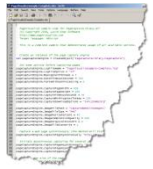
- By convention, constructors name are kept with a *capitalized name*



Function Objects

- Function objects inherit from *Function.prototype*
  - Modifications to the *Function.prototype* object are propagated to all Function instances

```
Person.prototype.newFunction = function() {
        console.log("Hi there");
}

johnDoe.newFunction(); // "Hi there"
eric.newFunction(); // "Hi there"
```

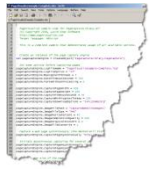


Enumerable

- Sometimes, you need to iterate over properties from an object
- You can do that thanks to *Object.keys(obj)*

```
var myObj = {
        myProp: 1,
        myMethod: function() { console.log("Plop"); }
}

Object.keys(myObj).forEach( function(key) {
        console.log(key + ": " + myObj[key]);
});
```

```
myProp: 1
myMethod: function () { console.log("Plop"); }
```



Enumerable

- *Object.keys(obj)*:
  - Returns an array of all *own* *enumerable* properties found upon a given object

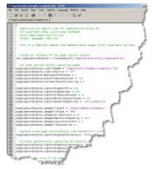
- You can also use *for-in* loop
  - Difference being that it enumerates properties in the prototype chain as well

Enumerable

- *for-in* example:

```
var myObj = {
        myProp: 1,
        myMethod: function() { console.log("Plop"); }
}
var myOtherObj = Object.create(myObj);
myOtherObj.myOtherProp = 2;

var propKey;
for(propKey in myOtherObj) {
        console.log(propKey + ": " + myOtherObj[propKey]);
}
```

```
myOtherProp: 2
myProp: 1
myMethod: function () { console.log("Plop"); }
```



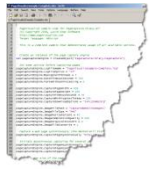
Private members

- But how to declare private members ?
  - Private members are just members only accessible by the object itself
  - So, an easy way to do that is to use variables limited to a restraint scope common with the other object members

Private members

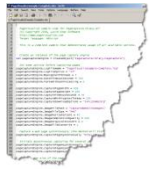
- Example with function objects :

```
function Foo() {
    var privateProperty = "private";
    function privateMethod() { ... };

    this.publicProperty = "public";
    this.publicMethod = function() { ... };
}
```



Introduction to closures

- This pattern of public or private members works because JavaScript has *Closures*
  - An inner function always has access to the vars and parameters of its outer function, even after the outer function has returned
  - Thanks to that, we can manipulate context bindings
    - We'll see that in the next chapter…

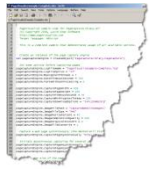


Introduction to closures

- Example of private shared property :

```
var myProto = {};

(function Scope() {
        var privateSharedProp = "private shared";

        myProto.publicProperty = "public";
        myProto.getPrivateSharedProp = function() {
                return privateSharedProp;
        }
}());

var obj = Object.create(myProto);
```

Questions ?

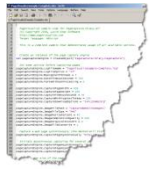


Exercise (1/3)

- The aim of the exercise will be to develop a simple web Tamagotchi

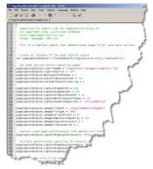
- Go on http://courses.supinfo.com and download the *Tamagocci - Base 1.zip* archive

- It contains a project with Jasmine tests



Exercise (2/3)

- A quick tour of the project :
  - *lib* folder: contains the Jasmine library
  - *spec* folder: contains the Jasmine tests
  - *src* folder: contains your JavaScript files
  - *SpecRunner.html* file: executes the tests and display a detailed report
  - *Tamagocci.html* file: a simple page using the library you will develop



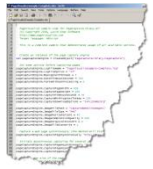
Exercise (3/3)

- For this first exercise:
  - You will write the code needed in *tamagocci.js* file to pass the tests

- When all the tests will be green, try to open *Tamagocci.html* into your browser

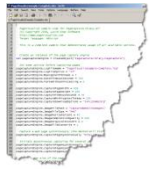
Go further with JavaScript…

CONTEXT BINDING



Presentation

- JavaScript provide three methods to bind default function context to another
  - Very useful to apply a function to a different scope
- We’re going to see them in that chapter…



Context binding

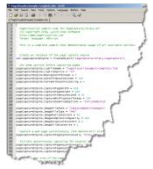
Apply

- *apply(…)* method allows to assign a different *this* object when calling an existing function
  - *this* refers to the current object or the calling context / scope

Apply - Example

```
var marion = {
    name: "Marion Cotillard",
    eat: function(foodType) {
        console.log(this.name+ " eat some " +foodType);
    },
    die: function() {
        console.log(this.name + " says: bweeeeeeuh...");
    }
};

var kevin = { name: "Kevin Doe" };

marion.eat.apply(kevin, ["beef"]);
marion.die.apply(kevin);
```

```
marion.eat.apply(kevin, ["beef"]);
marion.die.apply(kevin);
Kevin Doe eat some beef
Kevin Doe says: bweeeeeeuh...
```



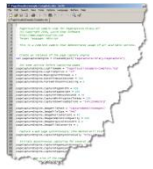
Context binding

Call

- *call(…)* do the same thing than *apply(…)*

- The difference is on the arguments
  - *apply(…)* accepts a single array of arguments
  - *call(…)* accepts an argument list

```
myObject.myFunction.apply(myOtherObject, [param1, param2]);

myObject.myFunction.call(myOtherObject, param1, param2);
```



Context binding

Bind

- *bind(…)* method creates a new function corresponding to the original…
  - … but with a different *this* !

```
anotherObject .getX();
42
>
```

```
var point = {
        x: 81, y: 18,
        getX: function() { return this.x; },
        getY: function() { return this.y; },
};
var anotherObject = { x: 42, ... };

anotherObject.getX = point.getX.bind(anotherObject);
anotherObject.getX(); // 42
```

Bind

- We can imagine that a simple version of the *bind()* method would be like this :

```
Function.prototype.bind = function bind(scope) {
        var self = this;
        return function() {
                return self.apply(scope, arguments);
        }
}
```

- But why define a *self* variable ?

Context binding

this

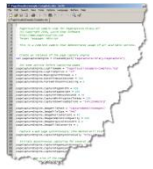
- As we said earlier, *this* represents the current context
- But in the returned method, the context is the *bind* function and not the prototype

```
Function.prototype.bind = function bind(scope) {
        var self = this; // this is the prototype context
        return function() {
                // Here, this != self because this is the bind
                // function context and not the prototype one.
                return self.apply(scope, arguments);
        }
}
```

Questions ?

Go further with JavaScript…

INHERITANCE



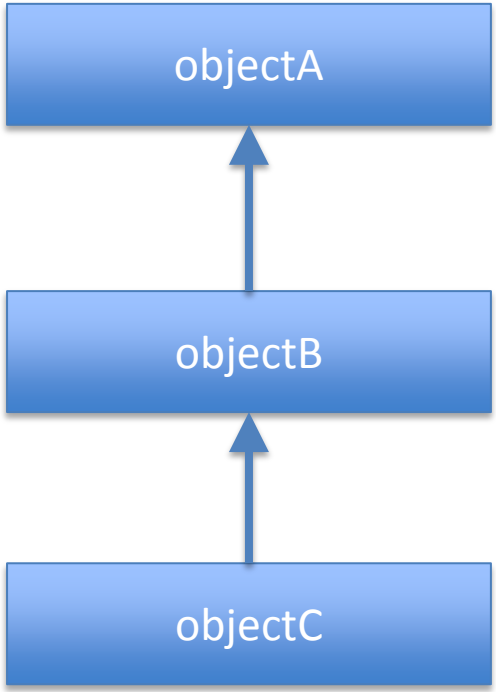
Presentation

- In Java or C#, inheritance is useful to:
  - Code reuse
  - Avoid the need for the programmer to write explicit casting operations (class hierarchy)
- JavaScript is a loosely typed language
  - No cast problem
- What matters about an object is what it can do!

Inheritance & Prototypes

- JavaScript is a prototypal language
  - Objects inherit directly from other objects called prototypes
  - An object can have and be a prototype at the same time
    - Like a class can be a sub-class and a parent-class

Prototype Chain

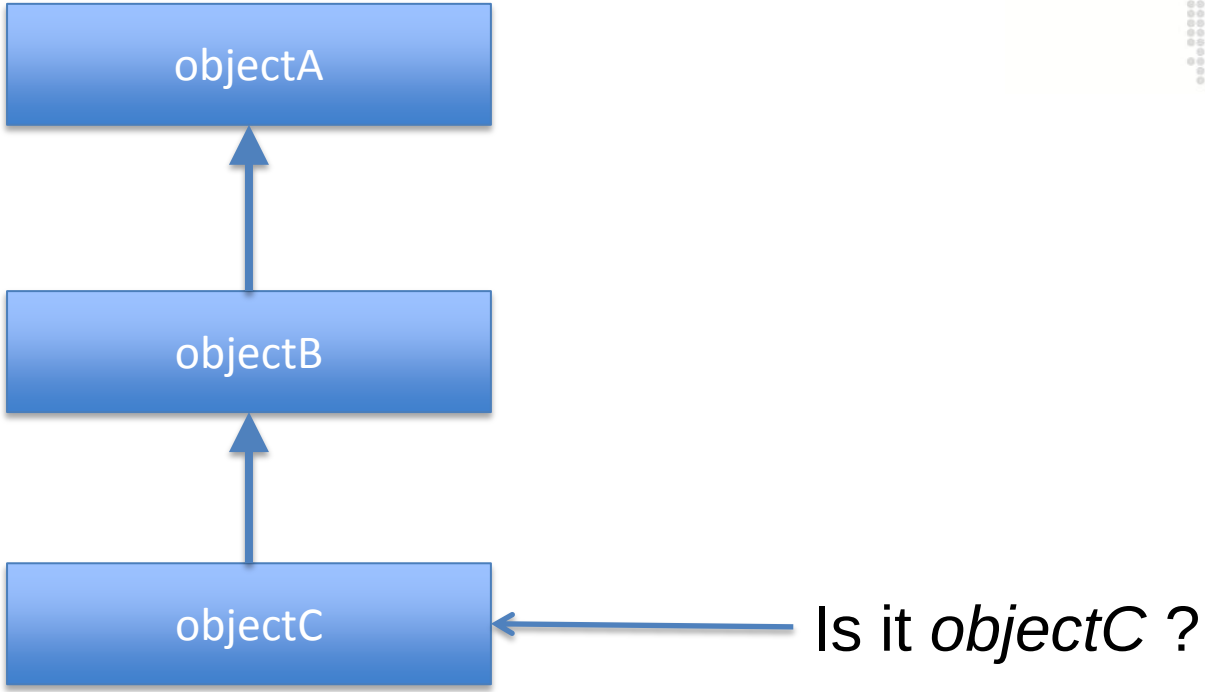



```
var objectA = { prop: 42 };

var objectB =
        Object.create(objectA);
objectB.anotherProp = 24;

var objectC =
        Object.create(objectB);

console.log(objectC.prop);
```

Who has *prop* ?

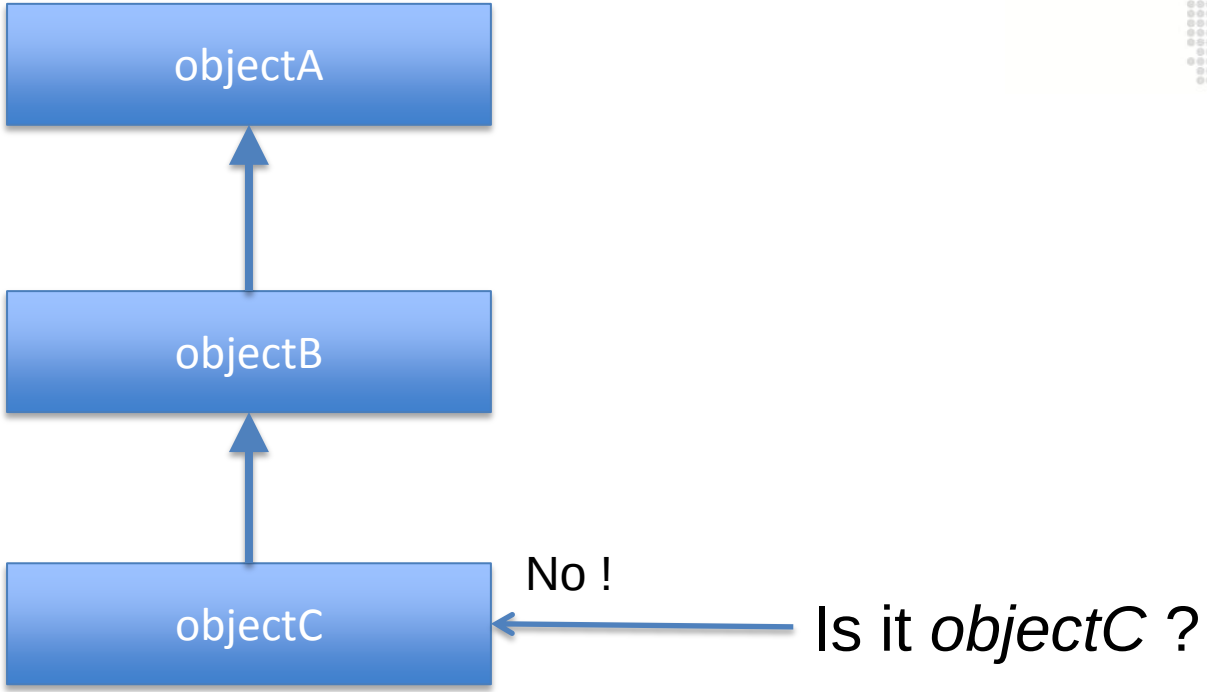
Inheritance

Prototype Chain



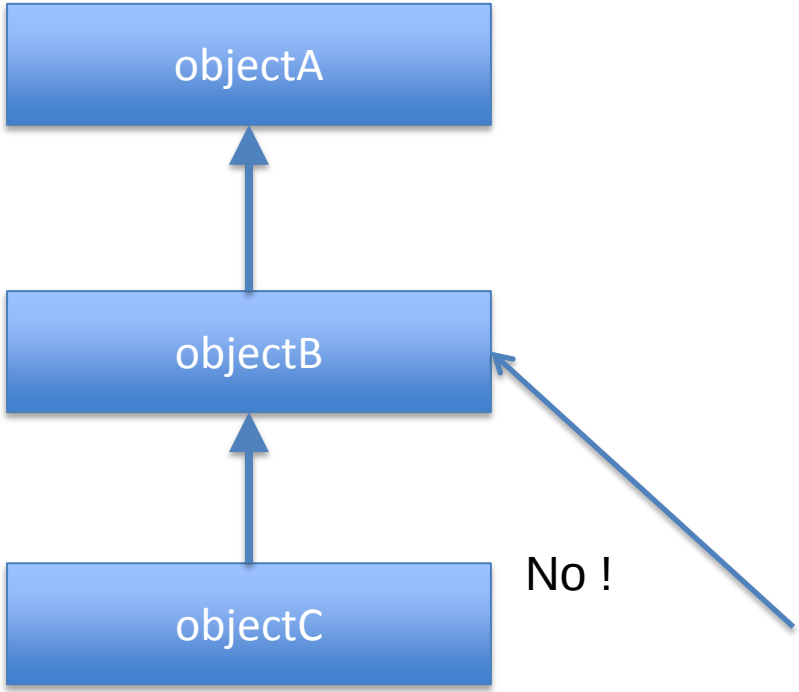
Inheritance

Prototype Chain



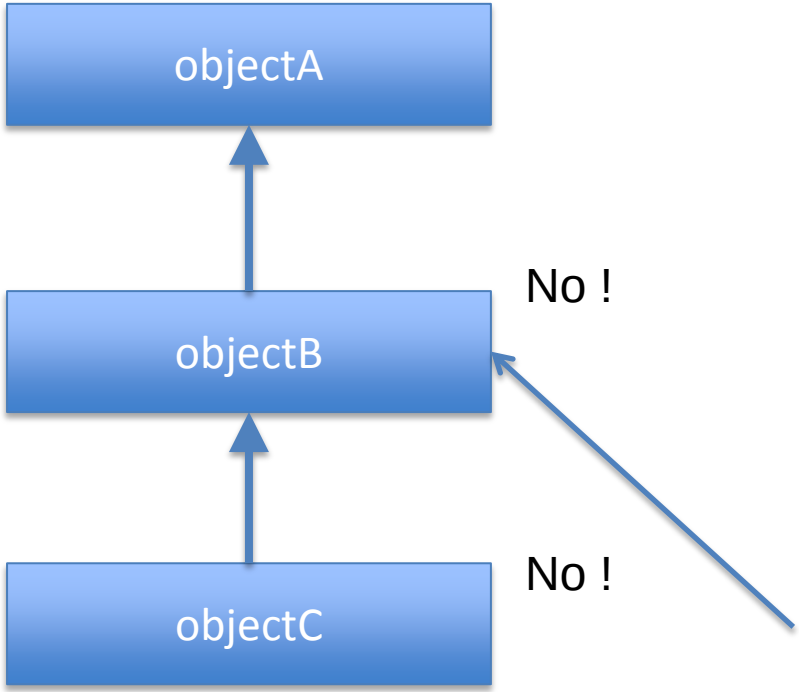
Inheritance

Prototype Chain



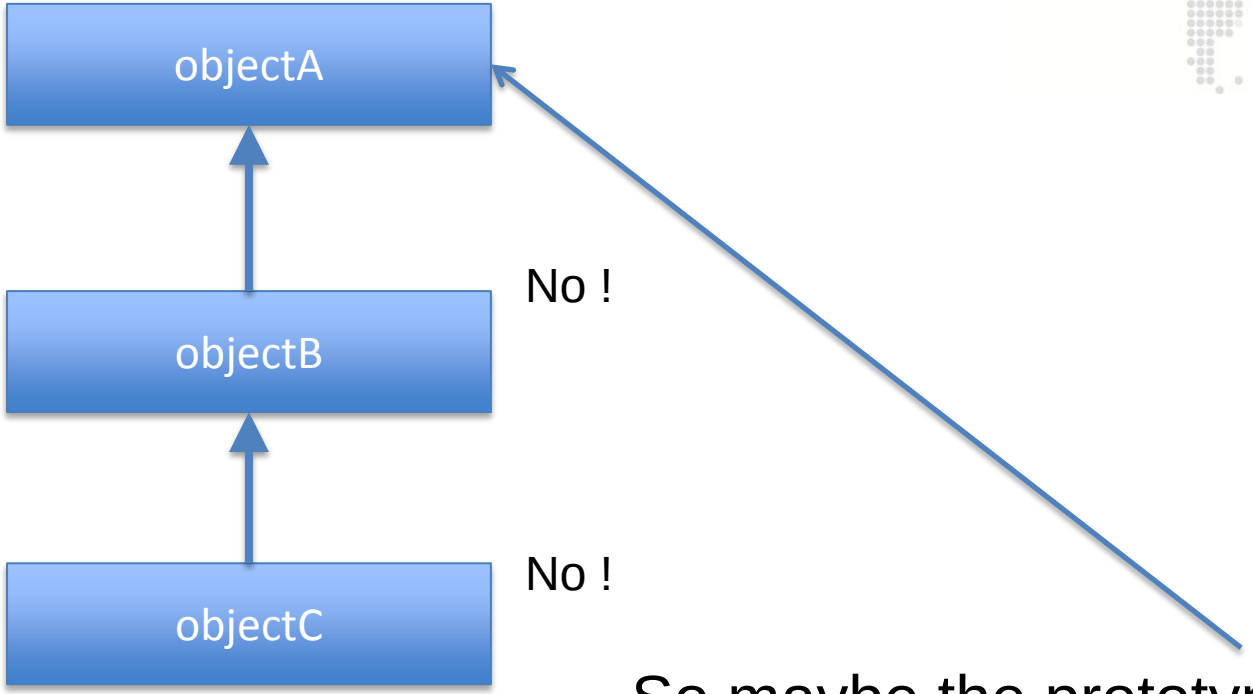
Inheritance

Prototype Chain



Inheritance

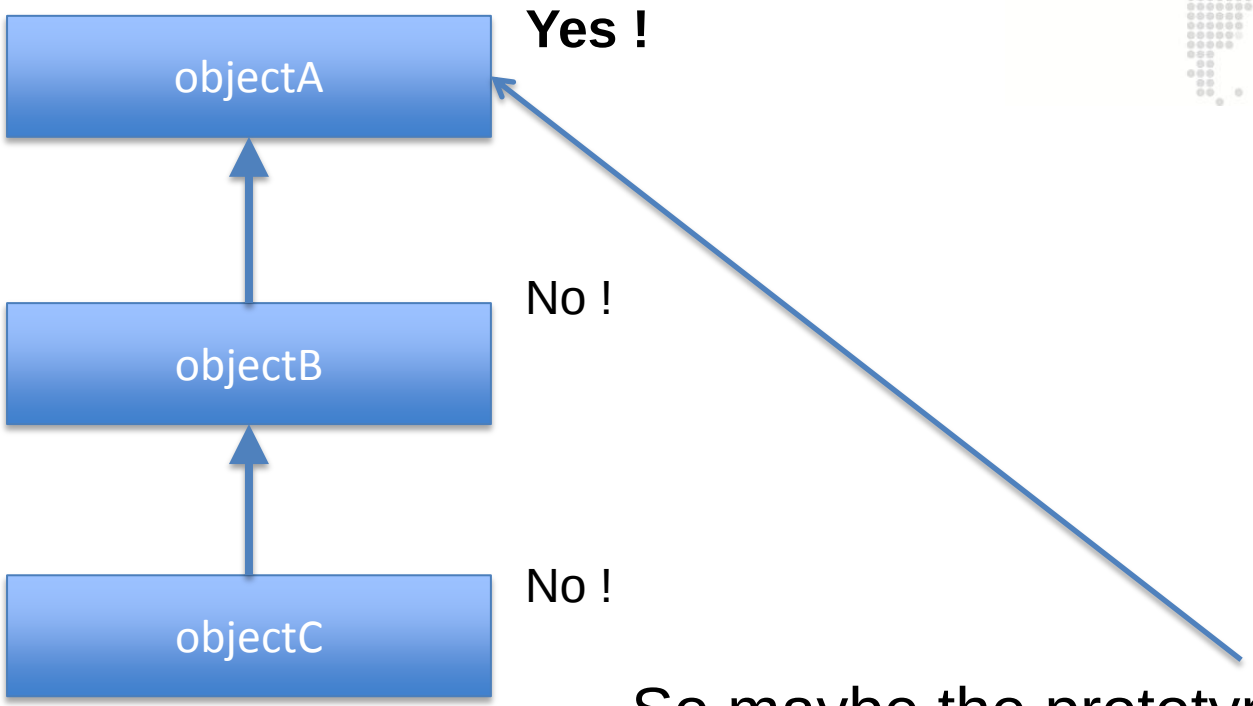
Prototype Chain



So maybe the prototype of the prototype ?

Inheritance

Prototype Chain



So maybe the prototype of the prototype ?

Inheritance with Object literals

- Example:

```
var phone = {
        dial: function(number) { ... }
}

var cellular = Object.create(phone);
cellular.sendSms = function(text) { ... };

var smartPhone = Object.create(cellular);
smartPhone.browseWebsite = function(url) { ... };
```

Inheritance with Function Objects

- Example:

```
function Person(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
};

Person.prototype.eat = function() {
        console.log("He/She is eating");
};

Person.prototype.sleep = function() {
        console.log("He/She is sleeping");
};
```

Inheritance with Function Objects

- Example:

```
function Employee(firstName, lastName, employeeId) {
        Person.apply(this, [firstName, lastName]);
        this.employeeId = employeeId;
}

Employee.prototype = Object.create(Person.prototype);

Employee.prototype.work = function() {
        console.log("He/She is working");
};
```

Inheritance with Function Objects

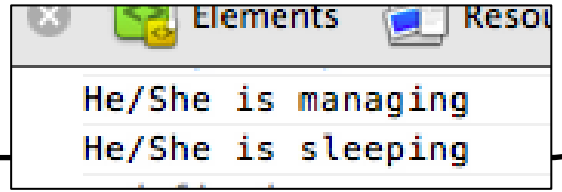
- Example:

```
function Manager(firstName, lastName, employeeId) {
    Employee.apply(this, [firstName, lastName, employeeId]);
}

Manager.prototype = Object.create(Employee.prototype);

Manager.prototype.work = function() {
       console.log("He/She is managing");
};
```

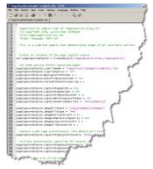
Inheritance with Function Objects

- Example:

```
var barney = new Manager("Barney", "Stinson", 1234);
barney.work();

Person.prototype.sleep = function() {
        console.log("He/She is sleeping");
};

barney.sleep();
```



Questions ?

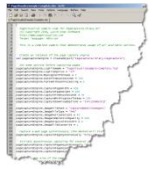


Exercise

- Go on http://courses.supinfo.com and download the *Tamagocci - Base 2.zip* archive
- Replace the *Tamagocci.js* by your own code
- Update it to make it pass the new tests

Go further with JavaScript…

PROPERTY DESCRIPTORS

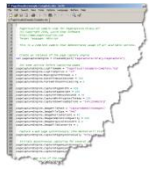


Presentation

- Each object property is linked to an object called **Property Descriptor**

- Two types of Property Descriptors:
  - Data descriptors
  - Accessor descriptors

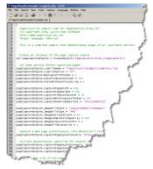
- We’re going to see both of them!



Data Descriptors

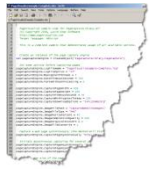
- A Data Descriptor is a property that has a value, which may or may not be writable

- A Data Descriptor is defined by an object with the fields described in the next slide…



Data Descriptors

Field	Explanation	Default value
value	The value associated with the property.	*undefined*
writable	True if and only if the value associated with the property may be changed.	*false*
configurable	True if and only if the type of this property descriptor may be changed and if the property may be deleted from the corresponding object.	*false*
enumerable	True if and only if this property shows up during enumeration of the properties on the corresponding object.	*false*



Data Descriptors

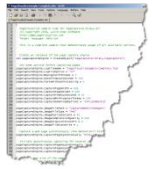
- Example:

```
var obj = {};
Object.defineProperty(obj, "myProp", {
        value: 42,
        writable: true,
        enumerable: true,
        configurable: true
});

obj.myProp = 51;
console.log(obj.myProp); // 51
console.log(Object.keys(obj)); // ["myProp"]
console.log(delete obj.myProp); // true
```

Data Descriptors

- Example:

```
var obj = {};
Object.defineProperty(obj, "myProp", {
        value: 42,
        writable: false,
        enumerable: false,
        configurable: false
});

obj.myProp = 51; // Didn't work. No error thrown.
console.log(obj.myProp); // 42
console.log(Object.keys(obj)); // []
console.log(delete obj.myProp); // false
```



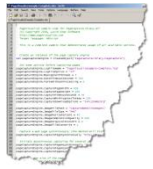
Accessor Descriptors

- An Accessor Descriptor is a property described by a getter-setter pair of functions

- An Accessor Descriptor is defined by an object with the fields described in the next slide…

Accessor Descriptors

Field	Explanation	Default value
get	A function which serves as a getter for the property, or undefined if there is no getter.	*undefined*
set	A function which serves as a setter for the property, or undefined if there is no setter.	*undefined*
configurable	True if and only if the type of this property descriptor may be changed and if the property may be deleted from the corresponding object.	*false*
enumerable	True if and only if this property shows up during enumeration of the properties on the corresponding object.	*false*



Accessor Descriptors

- Example:

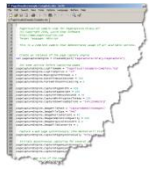
```
var obj = {}, ageValue;
Object.defineProperty(obj, "age", {
        get: function() { return ageValue; },
        set: function(newValue) {
                if(newValue >= 0) ageValue = newValue;
        }
});

obj.age = 12;
console.log(obj.age); // 12
obj.age = -1;
console.log(obj.age); // 12
```

Several ways to create a property…

- Be careful:

```
obj.prop = 1;

// is equivalent to:
Object.defineProperty(obj,
"prop", {
        value : 1,
        writable : true,
        configurable : true,
        enumerable : true
});
```

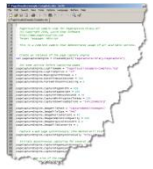
```
Object.defineProperty(obj,
"prop", { value : 1 });

// is equivalent to:
Object.defineProperty(obj,
"prop", {
        value : 1,
        writable : false,
        configurable : false,
        enumerable : false
});
```



Retrieve a Property Descriptor

- To retrieve a property descriptor for an own property:
  - You can use *Object.getOwnPropertyDescriptor(…)*

```
var obj = {};
Object.defineProperty(obj, "myProp", { value : 1 });


var desc = Object.getOwnPropertyDescriptor(obj, "myProp");
```



Define several descriptors

- Another useful method is :

  Object.defineProperties(…)

- Allows you to define or modify several properties at once

- Very useful to define an object structure

Define several descriptors

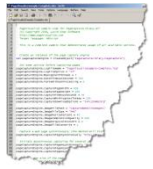
- Example :

```
var jack = Object.defineProperties({}, {
        firstName: { value: "Jack" },
        lastName: { value: "Harkness" },
        age: { value: 45, writable: true }
});
```

Define several descriptors

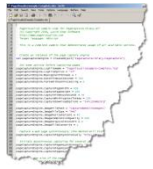
- Example :

```
var Person = function(firstName, lastName, age) {
        Object.defineProperties(this, {
                firstName: { value: firstName },
                lastName: { value: lastName },
                age: { value: age },
        });
};

var jack = new Person("Jack", "Harkness", 45);
```



Frozen & Sealed objects

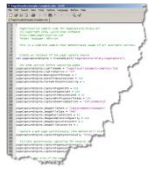
- You can seal an object to:
  - prevent new properties from being added to it
  - mark all existing properties as non-configurable



Frozen & Sealed objects

- Example:

```
var obj = { foo: "bar" };
Object.seal(obj);

console.log(Object.isSealed(obj)); // true
obj.foo = "foo"; // still works
obj.bar = "bar"; // silently doesn't add the property
Object.defineProperty(obj, "foo", {
        get: function() { return "g"; }
}); // throws a TypeError
```



Frozen & Sealed objects

- You can freeze an object to prevent:
  - new properties from being added to it
  - existing properties from being removed
  - existing properties, or their enumerability, configurability, or writability, from being changed

Frozen & Sealed objects

- Example:

```
var obj = { foo: "bar" };
Object.freeze(obj);

console.log(Object.isFrozen(obj)); // true
obj.foo = "foo"; // silently does nothing
obj.bar = "bar"; // silently doesn't add the property
```

Questions ?

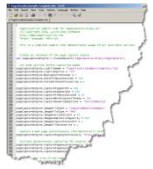
Go further with JavaScript…

STRICT MODE

Presentation

- ECMAScript 5 introduce *Strict mode*

- A way to opt in to a restricted variant of JavaScript



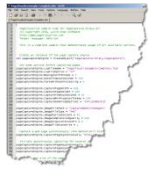
What is the difference ?

- Several changes to normal semantics:
  - Eliminates some language pitfalls that didn't cause errors by changing them to produce errors
  - Fixes mistakes that make it difficult for JS engines to perform optimizations
  - Prohibits some syntax likely to be defined in future versions of ECMAScript

Scope

- Strict mode applies to entire scripts or to individual functions

- It doesn't apply to block statements
  - if, for, while…



Put the strict mode on

- To invoke strict mode for an entire script
  - Put "use strict" before any other statements

```
"use strict";
var name = prompt("What is your name ?");
console.log("Hi " + name + "! I'm in strict mode baby.");
```

Put the strict mode on

- To invoke strict mode for functions :
  - Put "use strict" in the function body before any other statements

```
function strictFunction() {
   "use strict";
   var name = prompt("What is your name ?");
   console.log("Hi " + name + "! It's strict mode baby.");
}
function notStrictFunction() {
   var name = prompt("What is your name ?");
   console.log("Hi " + name + "!");
}
```

Converting mistakes into errors

- Strict mode makes it impossible to accidentally create global variables:

```
"use strict";
unknownVariable = 42; // throws a ReferenceError
```

- Strict mode makes assignments which would otherwise silently fail throw an exception:

```
"use strict";
var obj1 = {};
Object.defineProperty(obj1, "x", { value: 42 }); // read only
obj1.x = 9; // throws a TypeError
```

Paving the way for the future…

- In strict mode a short list of identifiers become reserved keywords:

```
function package(protected) { // !!!
  "use strict";
  var implements; // !!!
  interface: // !!!
  while (true)
  {
    break interface; // !!!
  }
  function private() { } // !!!
}
```

Paving the way for the future…

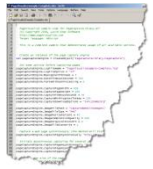
- Strict mode prohibits function statements not at the top level of a script or function:

```
"use strict";
if (true) {
  function myFunction() { return 1; } // !!! syntax error
  myFunction();
}

for (var i = 0; i < 5; i += 1) {
  function myFunction() { return 2; } // !!! syntax error
  myFunction();
}
```

Questions ?

Go further with JavaScript…

GOOD PRACTICES



Global variables

- The language makes it easy to define…
- … but excessive uses weaken the resiliency of programs

- A good practice to minimize the use of global variables
  - Create a single global variable for your app!

Global variables

- Example:

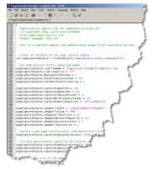
```
var MY_SUPER_APP = {};

MY_SUPER_APP.myVar = 12;
MY_SUPER_APP.myFunction = function() { ... };
MY_SUPER_APP.myObject = { ... };
```

- Only *MY_SUPER_APP* is global

Global variables

- Another way is to declare all your code inside a function you invoke immediately:

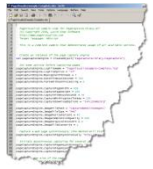
```
(function() {
        var myVar = 12;
        var myObject = { ... };

        function createArticle() {
                ...
        }
})();
```



Global variables

- Why global variables are dangerous?
  - Because it can cause variable names collisions!
  - Bad interactions with other apps, widgets or libraries

- Java propose packages
- C# propose namespaces
- In JavaScript, you can easily use composition !



Global variables

- There is another way to avoid that problem:
  - **Make your code modular !**
- We'll see more about that in the next chapter…

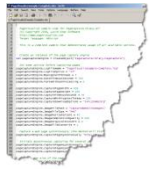
Give a name to Fn expressions

- In most of previous examples, they're without name…
- … but name them bring some advantages :
  - Auto-documentation: we know what they do
  - Easier to debug: more readable stacktraces

Questions ?

Go further with JavaScript…

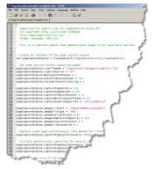
MODULAR JAVASCRIPT



Modularity

- A modular application is composed of a set of highly decoupled, distinct pieces of functionality stored in modules

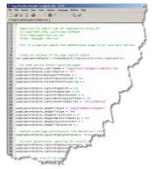
- Do you remember of loose coupling ?
  - Limit the dependencies to keep an easier maintainability



Modularity

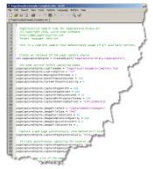
- JavaScript applications are more and more consequent
  - Need to be more organized

- For now, ECMAScript doesn't provide a standard specification for that
  - But planned for ECMAScript 6…



Modularity

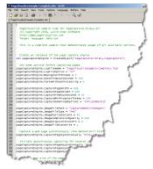
- Several non standardized formats for writing modular JavaScript exists
  - CommonJS
  - AMD
  - …
- We’re going to focus on CommonJS



CommonJS

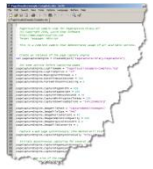
- Project with the goal of specifying an ecosystem for JavaScript outside the browser
  - Started in 2009 and initially named ServerJS

- CommonJS module proposal specifies a simple API for declaring modules server-side



CommonJS

- CJS modules are reusable piece of JavaScript which exports specific objects…
- … made available to any dependent code !
- We’re going to see how to define and access to a module in the next slides…



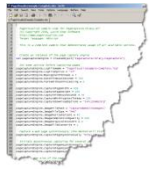
CommonJS - Define a module

- CommonJS introduce a variable named *exports*
- It contains the objects a module wishes to make available to other modules

CommonJS - Define a module

- Example (*my_module.js*) :

```
var Foo = function() {
        this.property = 42;
}

function bar() {
        console.log("bar!");
}

exports.FooObject=Foo;
exports.bar=bar;
```



CommonJS - Access to a module

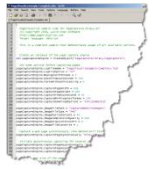
- To access a module, CommonJS introduce a *require(…)* function

- It takes in argument the path to the module file and return an object containing all its exported objects

CommonJS - Access to a module

- Example (*app.js*) :

```
// Import the CommonJS module.
// The argument is the module file path without extension
var myModule = require("my_module");

var myFoo = new myModule.FooObject();
myModule.bar(); // bar!
```

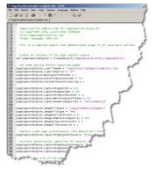


CommonJS - Advantages

- This system provide a good code isolation
  - More abstraction !
    - Only explicit objects of the module are exposed (or exported)
  - Good protection against name collisions !
    - *require()* return an object representing the module and containing its exposed APIs

Who implement the spec ?

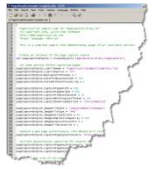
- The CommonJS specification is implemented by a lot of products like :
  - JSBuild
  - RequireJS
  - curl.js
  - node.js



Why to see CommonJS ?

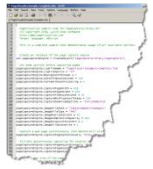
- CommonJS VS AMD ?
  - AMD is a very good solution with browser-friendly features
  - CommonJS is perfect in a server environment and used by NodeJS ad Wakanda
- We need to see CommonJS to understand next chapters about Server-side JavaScript…

Questions ?



Exercise

- Go on http://courses.supinfo.com and download the *Tamagocci - Base 3.zip* archive
- Replace the *Tamagocci.js* by your own code
- Update it to make it pass the new tests



Exercise (optional)

- Download require.js on http://requirejs.org/
- Refactor your Tamagocci.js to transform it into CommonJS module
- Use your new module thanks to Require.js !