# Advanced JavaScript

ITM - SDM S9

EPITA

## Strict Mode

`"use strict";` Defines that Javascript code should be executed in "strict mode"

## The `"use strict"` Directive

- The "use strict" directive was new in ECMAScript version 5.
- It is not a statement, but a literal expression, ignored by earlier versions of JavaScript.
- The purpose of "use strict" is to indicate that the code should be executed in "strict mode".
- With strict mode, you can not, for example, use undeclared variables.

## Declaring Strict Mode

Strict mode is declared by adding `"use strict";` to the beginning of a script or a function.

Declared at the beginning of a script, it has global scope (all code in the script will execute in strict mode):

```
"use strict";

x = 3.14; // This will cause an error because x is not declared
```

```
"use strict";
myFunction();

function myFunction () {
  y = 3.14; // This will also cause an error because y is not declared
}
```

Declared inside a function, it has local scope (only the code inside the function is in strict mode):

```
x = 3.14; // This will not cause an error.
myFunction();


function myFunction () {
  "use strict";

  y = 3.14; // This will cause an error.
}
```

## Why Strict Mode?

- Strict mode makes it easier to write "secure" JavaScript.
- Strict mode changes previously accepted "bad syntax" into real errors.
- As an example, in normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, making it impossible to accidentally create a global variable.
- In normal JavaScript, a developer will not receive any error feedback assigning values to non-writable properties.
- In strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.

```
var a = 2;

(function foo(){

    var a = 3;
    console.log( a ); // 3

})();

console.log( a ); // 2
```

Instead of treating the function as a standard declaration like in the previous snippet, the function is treated as **function-expression**.

`(function foo(){ .. })` as an expression means the identifier `foo` is found *only* in the scope where the `..` indicates, not in the outer scope. Hiding the name `foo` inside itself means it does not pollute the enclosing scope unnecessarily.

## Scope

## What is the scope?

**Scope is the set of rules that determines where and how a variable (identifier) can be looked-up**

## Three important entities :

1. ***Engine :*** charge of **compilation** and **execution** of JavaScript program.
2. ***Compiler*** : handles all the dirty work of parsing and code-generation.
3. ***Scope*** : ***collects and maintains a look-up list*** of all the declared variables, and enforces a strict set of rules as to how these are accessible to currently executing code.

## What happens when the compiler sees `var a = 2;` statement ?

In fact, there are two statements:

1. `var a` which **Compiler** handle during compilation
2. `a = 2` which **Engine** handle during exection

Compiler will instead proceed as:

1. **Compiler** asks to if variable a exists. If so, **Compiler** ignores this declaration and moves on. If not, **Compiler** asks to **Scope** to declare a new variable called `a` for that scope declaration.

2. **Compiler** produces code for **Engine**, to handle `a = 2`. The **Engine** begin to asks to **Scope** if there is a varibale called `a` in the current scope. If so, **Engine** uses variable. If not, Engine looks elsewhere

# Lexical Scope

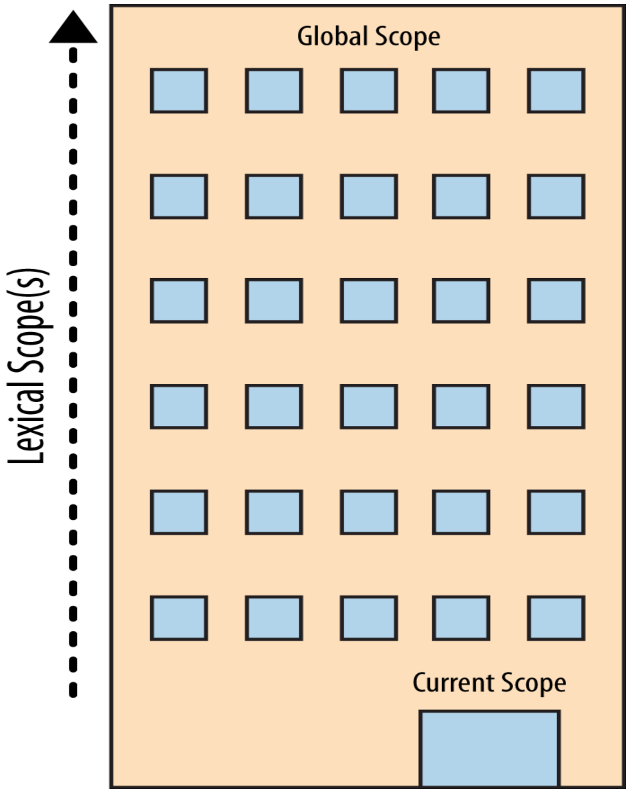
## What is the lexing ?

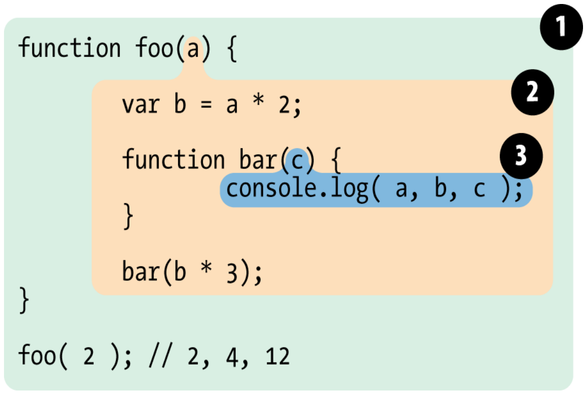
***The lexing process examines a string of source code characters and assigns semantic meaning to the tokens as a result of some stateful parsing.***

## What is the lexical scope ?

Lexical scope is based on where variables and blocks of scope are authored, by you, at write time.

## Nested Scope





```
function foo(a) {

        var b = a * 2;

        function bar(c) {
                console.log( a, b, c );
        }

        bar(b * 3);
}

foo( 2 ); // 2, 4, 12
```

**Bubble 1** covered the global scope which includes identifier: `foo`.

**Bubble 2** covered the scope of `foo` which includes identifiers: `bar`, `a` and `b`.

**Bubble 3** covered the scope of `bar` which includes identifier: `c`.

In this above code snippet, the ***Engine*** executes the `console.log(...)` and goes looking for the three referenced variables `a`, `b`, and `c`. It begins with innermost scope bubble, the scope of the `bar(..)` function. It won't find `a` there, so it goes up one level, out to the next nearest scope bubble, the scope of `foo(..)`. It finds `a` there, and so it uses that `a`. Same thing for `b`. `c` is found inside scope of `bar`.

## Function As Scopes

```
var a = 2;

function foo() {

    var a = 3;
    console.log( a ); // 3

}
foo();

console.log( a ); // 2
```

In this chunk of code, there are few problems:

- The first is that we have to declare a named-function foo(), which means that the identifier name foo itself "pollutes" the enclosing scope (global, in this case).

- We also have to explicitly call the function by name (foo()) so that the wrapped code actually executes.

## Hoisting

```
a = 2;

var a;

console.log( a );
```

What do you expect ?

## And in this piece of code ?

```
console.log( a );

var a = 2;
```

## Why ?

**Variables and function are processed first before any part of your is executed**

```
a = 2;

var a;

console.log( a );
```

We see this.

```
/* Compilation time */

var a;


/* Execution time */

a = 2;

console.log( a );
```

JavaScript sees that.

```
console.log( a );

var a = 2;
```

We see this.

```
/* Compilation time */

var a;


/* Execution time */

console.log( a );

a = 2;
```

JavaScript sees that.

## It's also important to note that hoisting is per-scope

```
function foo() {
  var a;

  console.log( a ); // undefined

  a = 2;
}

foo();
```

The program is interpreted like this :

```
function foo() {
  var a;

  console.log( a ); // undefined

  a = 2;
}

foo();
```

## Function expressions are not hoisted only function declarations

```
foo(); // TypeError!

var foo = function bar() {
  // ...
};
```

## Function first

```
foo(); // 1

var foo;

function foo() {
  console.log( 1 );
}

foo = function() {
  console.log( 2 );
};
```

The snippet code is interpreted by JavaScript like this:

```
function foo() {
  console.log( 1 );
}

foo(); // 1

foo = function() {
  console.log( 2 );
};
```

## Closure

*Definition* :

> Closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope.

```
function foo() {
    var a = 2;

    function bar() { console.log( a ); }

    return bar;
}

var baz = foo();

baz(); // 2 -- Closure
```

The function has lexical scope access to the inner of `foo()`. But then, we take `bar()`, the function itself, and pass it **as** a value. In this case, we `return` the function object itself that `bar` references.

After we execute `foo()`, we assign the value it returned (our inner `bar()` function) to a variable called `baz`, and then we actually invoke `baz()`, which of course is invoking our inner function `bar()`, just by a different identifier reference.

```
for (var i=1; i<=3; i++) {
    setTimeout( function timer(){
        console.log("Print : " + i);
    }, i*1000 );
}
```

What is display on the JavaScript console ?

Print : 4

Print : 4

Print : 4

**Why ?**

The timeout function callbacks are running well after the completion loop.

**What do we want ?**

We want that each iteration of loop captures its own copy of variable i.

## IIFE(Immediately Invoked Function Expression)

```
for (var i=1; i<=3; i++) {
    (function(){
        var j = i;
        setTimeout( function timer(){
            console.log("Print : " + j);
        }, j*1000 );
    })();
}
```

With `let`:

```
for (let i=1; i<=3; i++) {
        setTimeout(function timer(){
            console.log("Print : " + j);
        }, j*1000 );
}
```

With `let` keyword variable will be declared not just once for the loop, **but each iteration.**

## Modules

```
function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
}

var foo = CoolModule();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

## Objects

## Syntax

Javascript object can be created by two forms:

1. Declarative form:

```
var myObj = {
  key: value
}
```

1. Constructed form:

```
var myObj = new Object();

myObj.key = value;
```

## Built-in Objects

There are several other object sub-types:

- `String`
- `Number`
- `Boolean`
- `Object`
- `Function`
- `Array`
- `Date`
- `RegExp`
- `Error`

This objects can be used as a constructor with a `new` operator:

```
var strOjbect = new String("I am a string")
```

## Object coercion

```
var strPrimitive = "I am string"; // this is a string primitive.
```

To perform operation on primitive type a built-in objects are required, in our case "String" if we want to realize operation like checking its length.

Fortunately, the language automatically coerces a `"string"` primitive to a `String` object for perform operation like checking length of string. So, you almost never need to use create Object form.

```
var strPrimitive = "I am string"; // this is a string primitive.
console.log( strPrimitive.lenght );      // 11
```

**Only use the constructed form if you need the extra options.**

## Content

```
var myObject = {
  a: 2
};

myObject.a;  //2

myObject[a]; //2
```

The engine JavaScript doesn't store the value of property but it stores the property name which act as pointers to where the values are stored.

To access the values of properties object, we can use either the `.` operator or the `[]` operator.

The main difference between two syntax is that `.` operator requires an `Identifier` compatible name after it, whereas the `[]` syntax can take basically any UTF-8/Unicocde compatible string as the name property.

```
var myObject = {
  "Weird-Key?": 2
};

myObject["Weird-Key?"];  // 2
```

With `[]`, we can programtically build up the value of string, such as:

```
  var wantA = true;
  var myObject = {
  a: 2
  };

  var idx;

  if (wantA) {
  idx = "a";
  }

  // later

  console.log( myObject[idx] ); // 2
```

With also can compute property names, like this:

```
var prefix = "foo";

var myObject = {
[prefix + "bar"]: "hello",
[prefix + "baz"]: "world"
};

myObject["foobar"]; // hello
myObject["foobaz"]; // world
```

# Arrays

In Javascript, arrays are objects that have some array-like characteristics.

## Array Literals

Array Literal is an pair of square brackets surrounding zero or more values separated by commas.

The first value will get the property name '0', the seconde value will get the property name '1' and so on.

```
var myArray = [];

var myNumbers = [
    'zero', 'one', 'two', 'three', 'four',
    'five', 'six', 'seven', 'height', 'nine'
];

console.log( myArray[1] ); // undefined
console.log( myNumbers[1] ); // one


console.log( myArray.length ); // 0
console.log( myNumbers.length ); // 10
```

But in Javascript we can get similar result with an object:

```
var myNumbersObject = {
'0': 'zero', '1': 'one', '2': 'two',
'3': 'three', '4': 'four', '5': 'five',
'6': 'six', '7': 'seven', '8': 'height',
'9': 'nine'
};

console.log(myNumbersObject['0']); // zero
```

But there are also significant diffrences. `Array` has a set of useful methods that `Object` does not have.

JavaScript allows an `array` to contain any mixtures of values:

```
var misc = [
  'string', 30, true, false, undefined,
  {propertyObject: '44'}, ['nested', 'array']
];

console.log( misc.length );
```

## Length

Javascript's array `length` is not an upper bound. If you store an element with a subscript that is greater than or equal to the current `length` will increase to contain the new element.

```
var myArray = [];
myArray.length;   // 0

myArray[1000000] = true;
myArray.length  // 1000001
```

The `length` of array can be set explicitly. Making `length` larger does not allocate more space for the array. Making the `length` will cause all properties with a subscript that is greater than or equal to the new length to be deleted:

```
var myNumbers = [
    'zero', 'one', 'two', 'three', 'four',
    'five', 'six', 'seven', 'height', 'nine'
];

myNumbers.length = 3;

// myNumbers is ['zero', 'one', 'two']
```

## Add

We can add new element with method `push` :

```
var myNumbers = [
    'zero', 'one', 'two', 'three', 'four',
    'five', 'six', 'seven', 'height', 'nine'
];

myNumbers.push('ten'); // ["zero", "one", "two", "three",
// "four", "five", "six", "seven", "height", "nine", "ten"]
```

## Delete

We can delete array with method `splice`:

```
var myNumbers = [
    'zero', 'one', 'two', 'three', 'four',
    'five', 'six', 'seven', 'height', 'nine'
];

myNumbers.splice(9, 1); // ["zero", "one", "two", "three",
// "four", "five", "six", "seven", "height"]
```

## Enumeration

We can enumerate array with a `for`:

```
var myNumbers = [
'zero', 'one', 'two', 'three', 'four',
'five', 'six', 'seven', 'height', 'nine'
];

for(var i = 0; i < myNumbers.length; i++) {
  console.log(i);
}
```

## Duplicating Objects:

With can deeply duplicate a object in Javascript with JSON or `Object.assign()`:

```
var myObject = {
    a: 'a',
    array: ['1', '2', '3', '4', '5'],
    object: {
        myFunction: function() {
            console.log('Hello');
        },
        string: 'Daddy'
    }
}

console.log(myObject);


var newMyObject = Object.assign({}, myObject)

var newMyObjectJSON = JSON.parse(JSON.stringify(newMyObject))

console.log(newMyObject);

console.log(newMyObjectJSON);
```

# Advanced JavaScript

ITM - SDM S9

EPITA

## Property Descriptor

In Javascript all properties are described with property descriptor

```
var myObject = {
    a: 4
};

Object.getOwnPropertyDescriptor( myObject, "a" );
/*
{
    value: 4,
    writable: true,
    enumerable: true,
    configurable: true
}
*/
```

`getOwnPropertyDescriptor` return property descriptor the property descriptor(data-descriptor) `value`, `writable`, `enumerable` and `configurable`

We can add new property or modify an existing one with method `defineProperty`.

## Writable

```
var myObject = {};

Object.defineProperty(myObject, "a", {
    value: 2,
    writable: false, // not writable
    configurable: true,
    enumerable: true
});

myObject.a = 3;

myObject.a; // 2
```

It does not raise error

But if you try this in strict mode we get an error

```
'use strict'

var myObject = {};

Object.defineProperty(myObject, "a", {
    value: 2,
    writable: false, // not writable
    configurable: true,
    enumerable: true
});

myObject.a = 3; // TypeError
```

## Configurable

As long as property is configurable, we can modify

```
var myObject = {
    a: 2
};

myObject.a = 3;
myObject.a;       // 3

Object.defineProperty( myObject, "a", {
    value: 4,
    writable: true,
    configurable: false,    // not configurable!
    enumerable: true
} );

myObject.a;       // 4
myObject.a = 5;
myObject.a;       // 5

Object.defineProperty( myObject, "a", {
    value: 6,
    writable: true,
    configurable: true,
    enumerable: true
} ); // TypeError
```

The program raise a `TypeError` because we attempt to change the descriptor definition of a non-configurable property.

Be when we change configurable property to `false`, we cannot go back(change to true)

```
var myObject = {
    a: 2
};

myObject.a;                // 2
delete myObject.a;
myObject.a;                // undefined

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: true,
    configurable: false,
    enumerable: true
} );

myObject.a;                // 2
delete myObject.a;
myObject.a;                // 2
```

`myObject.a` does not remove because it's non-configurable property so delete failed silenty

**Enumarable**

We can prevent object to use `for..in` for example, in setting property `enumerable` to `false`

## Constant object

We can create object property that cannot be changed, redefined or deleted by setting property `writable: false` and `configurable: false`

```
var myObject = {};

Object.defineProperty( myObject, "FAVORITE_NUMBER", {
    value: 42,
    writable: false,
    configurable: false
} );

myObject["FAVORITE_NUMBER"] = 43;
myObject["FAVORITE_NUMBER"]; // 42
```

## Prevent Extension

If we want to create an object that cannot have new properties, we can use `Object.preventExtensions`

```
var myObject = {
    a: 2
};

Object.preventExtensions( myObject );

myObject.b = 3;
myObject.b; // undefined
```

In non-strict mode, the creation of `b` fails silenty.

In strict mode, it throws a `TypeError`.

## Seal & Freeze

We can create sealed object, this object cannot add property, cannot reconfigure or delete any existing property but it can still modify their values.

`Object.freeze` creates frozen object, it is the same behavior that `Object.seal()` but we cannot modify properties. This approch is the highest level of immutability.

## Getters and Setters

When we want to create or access property object Javascript performed two actions `[[Put]]` and `[[Get]]` respectively.

Let's see how to override `[[Get]]` operation for property object.

There are two ways for realise this:

With object literal syntax:

```
var myObject = {
    // define a getter for `a`
    get a() {
        return 2;
    }
};

myObject.a; // 2
```

And with explicit definition with method `defineProperty`:

```
Object.defineProperty(
    myObject,    // target
    "a",        // property name
    {            // descriptor
        // define a getter for `a`
        get: function(){ return 2 },

        // make sure `a` shows up as an object property
        enumerable: true
    }
);

myObject.a; // 2
```

If we try to set the value of `a` later, the set operation won't throw an error but will just silenty.

Let's see how to override `[[Put]]` operation for property object

```
var myObject = {
    // define a getter for `a`
    get a() {
        return this._a_;
    },

    // define a setter for `a`
    set a(val) {
        this._a_ = val * 2;
    }
};

myObject.a = 2;

myObject.a; // 4
```

When we use getter and setter **the data-descriptor value and writable are ignored**.

## Existence

How to check if property exist in object ?

We can use operator `in`:

```
var myObject = {
    a: 2
};

("a" in myObject);                // true
("b" in myObject);              // false
```

Or we can use method `hasOwnProperty`

```
myObject.hasOwnProperty( "a" );    // true
myObject.hasOwnProperty( "b" );    // false
```

Difference between method `hasOwnProperty` and operator `in` is that operator `in` check if property exist in object or if it exist higher level of the [[Prototype]] chain object(we talk about protoype chain later) whereas method `hasOwnProperty` checks to see if only object has a property or not

## Enumeration

Go back to enumeration.

A bit earlier, we talk about date-descriptor `enumerable` which means that property object will be included if the object's properties are iterated through

```
var myObject = { };

Object.defineProperty(
    myObject,
    "a",
    // make `a` enumerable, as normal
    { enumerable: true, value: 2 }
);

Object.defineProperty(
    myObject,
    "b",
    // make `b` NON-enumerable
    { enumerable: false, value: 3 }
);

myObject.b; // 3
("b" in myObject); // true
myObject.hasOwnProperty( "b" ); // true

for (var k in myObject) {
    console.log( k, myObject[k] );
}
// "a" 2
```

Enumerable and non-emurable properties can be distinguished by another way:

```
var myObject = { };

Object.defineProperty(
    myObject,
    "a",
    // make `a` enumerable, as normal
    { enumerable: true, value: 2 }
);

Object.defineProperty(
    myObject,
    "b",
    // make `b` non-enumerable
    { enumerable: false, value: 3 }
);

myObject.propertyIsEnumerable( "a" ); // true
myObject.propertyIsEnumerable( "b" ); // false

Object.keys( myObject ); // ["a"]
Object.getOwnPropertyNames( myObject ); // ["a", "b"]
```

Method `propertyIsEnumarable` check if the given property name exist on the object and is also `enumerable:true`

`Object.keys()` returns an array of all enumerable properties whereas `Object.getOwnPropertyNames()` returns an array of all properties, enumerable or not.

## THIS

To answer a question: what this `this` a reference to ? We must to determine the call-site: the place in code where a function is called

We must determine which of 4 rules applies I'm going to explain this 4 rules.

## First rules: standalone function invocation

```
function foo() {
    console.log( this.a );
}

var a = 2;

foo(); // 2
```

When `foo()` is called, `this.a` resolves to our global variable a because in this case the default binding for this applies to the function call, and so points this at the global object.

Standalone rules is apply when the call-site of function is located with a plain, un-decorated function reference

## Implicit Binding

Second rule to consider is: does the call-site have context object

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

obj.foo(); // 2
```

The call-site uses the `obj` context to reference the function. The call of `foo()` is preceded by an object reference to `obj`. When there is a context object for function reference. The implicit binding rule says that it's that object which should be used for the function call's this binding.

## The last level of an object property reference chain matters to the call-site

```
function foo() {
    console.log( this.a );
}

var obj2 = {
    a: 42,
    foo: foo
};

var obj1 = {
    a: 2,
    obj2: obj2
};

obj1.obj2.foo(); // 42
```

## Implicit lost

Sometimes,the implicity bound function loses this binding which usually means it falls back to the default binding, of either the global object or undefined, depending on strict mode.

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var bar = obj.foo;  // function reference/alias!

var a = "oops, global"; // `a` also property on global object

bar(); // "oops, global"
```

Even though `bar` appears to be a reference to `obj.foo`, in fact, it's really just another reference to `foo` itself. Moreover, the call-site is what matters, and the call-site is `bar()`, which is a plain, un-decorated call and thus the default binding applies.

## Explicit Binding

What if you want to force function call to use a particular object for `this` binding, without putting a property function reference on a object ?

To realise this we can use to function `call(...)` and `apply(...)`. They both take, as their first parameter, an object to use for the `this`, and then invoke the function with that `this` specified

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2
};

foo.call( obj ); // 2
```

Invoking `foo` with explicit binding by `foo.call(..)` allows us to force its this to be obj.

## New binding

In Javascript `constructors` are just a functions that happpen to be called with the `new` in front of them. They are not attached to classes, nor are they instantiating a class. They are not even special types of functions. They're just regular functions that are, in essence, hijacked by the use of new in their invocation.

When a function is invoked with new in front of it, otherwise known as a constructor call, the following things are done automatically:

1. a brand new object is created (aka, constructed) out of thin air
2. the newly constructed object is [[Prototype]]-linked
3. the newly constructed object is set as the this binding for that function call
4. unless the function returns its own alternate object, the new-invoked function call will automatically return the newly constructed object.

```
function foo(a) {
    this.a = a;
}

var bar = new foo( 2 );
console.log( bar.a ); // 2
```

By calling foo(..) with new in front of it, we've constructed a new object and set that new object as the this for the call of foo(..). So new is the final way that a function call's this can be bound. We'll call this **new binding**.

## Mixins

In Javascript, there are no "classes" to instantiate, only object And objects are copied to other object, they are connected together

Other languages class behaviors imply copies.

So JS developpers **fake** the missing copy behavior of Javascript with **mixins**.

There are two types of mixin : **explicit** and **implicit**

## Explicit Mixins

We are going to study explicit mixin trough the example: Vehicle and Car

```
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        // only copy if not already present
        if (!(key in targetObj)) {
            targetObj[key] = sourceObj[key];
        }
    }

    return targetObj;
}

var Vehicle = {
    engines: 1,

    ignition: function() {
        console.log( "Turning on my engine." );
    },

    drive: function() {
        this.ignition();
        console.log( "Steering and moving forward!" );
    }
};

var Car = mixin( Vehicle, {
    wheels: 4,

    drive: function() {
        Vehicle.drive.call( this );
        console.log( "Rolling on all " + this.wheels + " wheels!" );
    }
} );
```

We only work object because **there are no class in Javascript** `Car` now has a copy of the properties and function from `Vehicle`

## Polymorphism Revisited

Let's see this statement: `Vehicle.drive.call( this )`. This is what call "explicit pseudo-polymorphism"

If we said `Vehicle.drive()`, the `this` binding for that function call would be the `Vehicle` object instead of the `Car`. So, we use `.call(this)`

Because of Javascript characteristic, explicit pseudo-polymorphism creates brittle manual/explicit linkage **in every single function where you need such a (pseudo-) polymorphic reference**.

Pseudo-polymorphic reference is harder to read and harder to maintain code, so ***Explicit pseudo-polymorphism should be avoided wherever possible***

## Implicit Mixins

```
var Something = {
    cool: function() {
        this.greeting = "Hello World";
        this.count = this.count ? this.count + 1 : 1;
    }
};

Something.cool();
Something.greeting; // "Hello World"
Something.count; // 1

var Another = {
    cool: function() {
        // implicit mixin of `Something` to `Another`
        Something.cool.call( this );
    }
};

Another.cool();
Another.greeting; // "Hello World"
Another.count; // 1 (not shared state with `Something`)
```

`Something.cool.call( this );` call in method `cool` of `Another` "borrow" the function `Something.cool()`and call it in the context of `Another` via `this` binding.

## Prototype

All Objects have an internal property denoted [[Prototype]]. It s simply a reference to another object. It is non-null at time of creation.

What's is the [[Prototype]] reference for? Operation `[[Get]]` is invoked when reference a property on an object.

First time, `[[Get]]` checks if object itself has property on it and if yes, it's used

But if not, [[Get]] operation look up in [[Prototype]] link of the object

```
var anotherObject = {
    a: 2
};

// create an object linked to `anotherObject`
var myObject = Object.create( anotherObject );

myObject.a; // 2
```

`Object.create(...)` creates an object with the [[Prototype]] linkage

But if `property` weren't found on `anotherObject`, the [[Prototype]] chain is again consulted and followed

If no matching property is ever found by the end of the chain, [[Get]] return `undefined`

## Where does the `[[Prototype]]` chain finish ?

[[Prototype]] chain finishes by the built-in `Object.prototype`.

This object includes common utilities like used by all Javascript objects like `.toString()`, `.valueOf()` and more.