Objects and prototypes

Create an object literal:
var cat = {name: 'Fluffy', color: 'white'};
display (cat.name);
Then we can add any property we want simply like that:
cat.age = 3
We can also add a function to that object:
cat.speak = function() { display("Meoowww") };

It could have been added initially in the cat definition:
var cat = {
name: 'Fluffy',
color: 'white',
speak: function () { display("Meoowww") };
};

Javascript allow a lot of freedom when working with object.

You create a class like in static languages (javascript is a dynamic language) with a constructor:
function Cat() { this.name = 'Fluffy',
this.color = 'White' }
var cat = new Cat();
This is a normal function. There is nothing special about that function.
By default this is the global object. In a web browser, this is the window object.
"new" creates a new empty javascript object, and set the context of "this" to the new object.

if you where to do the following: Then it sets 'White' to the window object.

```
function Cat() {
  // this.name = 'Fluffy'
  this.color = 'White'
}

var cat = Cat();

display(window.color)
```

White

Don't do that with 'name' as changing the window name will freak out the browser.

Create an object with a **constructor function** would be more general:
function Cat(name, colour) { this.name = name,
this.colour = colour }
This below is what happen when we write a constructor: **object.create**

```
var cat = Object.create(Object.prototype,
  {
    name: {
      value: 'Fluffy',
      enumerable:true,
      writable:true,
      configurable:true
    },
    color: {
      value:'White',
      enumerable:true,
      writable:true,
      configurable:true
    }
  })
```

Object properties:

Bracket notation: you do this cat['colour'] instead of this cat.colour
This is useful when the property is entered by the user input for example.

Property descriptor:

```
'use strict';

var cat = {
    name: 'Fluffy',
    color: 'White'
}

display(Object.getOwnPropertyDescriptor(cat, 'name'))
```

```
Object {
   value: Fluffy
   writable: true
   enumerable: true
   configurable: true
}
```

Writable attribute:
allow the property to be changed or not (public / private?) This will now throw an error if you try to change it (well only in strict mode, it would silently fail).

```
'use strict';

var cat = {
    name: {first: 'Fluffy', last: 'LaBeouf'},
    color: 'White'
}

Object.defineProperty(cat, 'name', {writable: false})
cat.name.first = 'Scratchy'
display(cat.name)
```

```
Object {
   first: Scratchy
   last: LaBeouf
}
```

Here even though 'name' is **read only** I can still change 'first' because the value of 'name' is an object and I can still change things in this object.
The way around that would be to write:
Object.**freeze**(cat.name); now the object for name can't be change

Enumerable attribute:
If set to false, then the property won't appear in for loop on the properties, or when returning the object keys.
display(Object.keys(cat))
This also affect the JSON serialisation of the object

Configurable attribute:
Lock down a property attributes from being changed (so affect value, enumerable but also configurable!)
You can still change 'writable'
But you can't delete the property: delete cat.name would throw an error.

Getters and setters:

```
'use strict';

var cat = {
    name: {first: 'Fluffy', last: 'LaBeouf'},
    color: 'White'
}

Object.defineProperty(cat, 'fullName',
  {
    get: function() {
      return this.name.first + ' ' + this.name.last
    }
  })

display(cat.fullName)
```

```
Fluffy LaBeouf
```

We create a function to get the full name of the cat. That's a possible getter.

Here is a possible setter:

```
'use strict';

var cat = {
    name: {first: 'Fluffy', last: 'LaBeouf'},
    color: 'White'
}

Object.defineProperty(cat, 'fullName',
  {
    get: function() {
      return this.name.first + ' ' + this.name.last
    },
    set: function(value) {
      var nameParts = value.split(' ')
      this.name.first = nameParts[0]
      this.name.last = nameParts[1]
    }
  })

cat.fullName = 'Muffin Top'
display(cat.fullName)
display(cat.name.first)
display(cat.name.last)
```

```
Muffin Top

Muffin

Top
```

Javascript Prototypes and inheritance

When there is a property missing from an object, we can create it. Here to get the last element of an array:

```
'use strict';

var arr = ['red', 'blue', 'green']

Object.defineProperty(arr, 'last', {get: function() {
  return this[this.length-1]
}})
var last = arr.last
display(last)
```

To make this available to all arrays:
Object.defineProperty(Array.prototype, 'last', {get: function () { return this[this.length-1] } })

A prototype is an object that exist on every function in JS.
Object do not have a prototype , they have a _proto_

```
display(cat.__proto__)
```

A function's prototype: A function's prototype is the object **instance** that will become the prototype for all objects created using this function as a constructor.

An object's prototype: An object's prototype is the object **instance** from which the object is inherited.

```
'use strict';

function Cat(name, color) {
  this.name = name
  this.color = color
}
var fluffy = new Cat('Fluffy', 'White')

display(Cat.prototype)
display(fluffy.__proto__)
display(Cat.prototype === fluffy.__proto__)
```

```
Cat {
}

Cat {
}

true
```

when using a property for an object, JS looks to see if the object has it's own property, if it doesn't, then it looks for its prototype.

object always inherit from another prototype, you can go back up the chain like that:

display (fluffy._proto_)
then display (fluffy._proto_._proto_) etc... untill a 'null' will be displayed
Object is the root and has a 'null' prototype.

Create our own inheritance:

```
'use strict';

function Animal() {
}
Animal.prototype.speak = function() {
  display('Grunt')
}

function Cat(name, color) {
  this.name = name
  this.color = color
}
Cat.prototype = Object.create(Animal.prototype)

var fluffy = new Cat('Fluffy', 'White')

fluffy.speak()
```

```
Grunt
```