capitalized first letter function name as constructor

"method " function against object i.e x.f()
f.apply(obj)
no object f(){this.name} //this refer to glaboal object

this. could refer to global object

What's the meaning of `this`?

Javascript's objects and functions

function as object,constructor,function

func.prototype --> constructor

func.__proto__ --> object

func() --> function

()=>{} arrow function : 1. can not use new constructor
2.this in function refer to upper closure object

COMP 2406

Here we look at javascript objects and functions as objects

The relationship between Objects and their Prototypes

Including especially:
Functions as objects and their capabilities and prototypes

The ability for any function to act as a constructor.

The idea that in javascript objects can have function properties but that these are not necessarily "methods".

Object-Oriented vs. Object-Based

Object-Oriented: Java, C++, C#, …

- There are objects and classes
- Classes are types and objects are memory of a particular type.
- Classes describe what properties all objects of that type have.
- Classes are code, objects are memory
- Inheritance is type-based: classes inherit from other classes
- Inheritance is typically an "is a" relationship

Object-Based: Javascript

- There are only objects, no classes (ES6 adds classes –sort of).
- Objects are created as individuals and need not be related to any type.
- "Inheritance" is object-based. An object can inherit from another object. (but in JS inheritance is just objects pointing to other objects –lots of confusion here.)
- New kinds of objects can be created without defining a type.
- Inheritance is not necessarily an "is a" relationship

BE FOREWARNED

- Javascript descriptions borrow words from OO languages but change their meaning or interpretation.
- Examples: `this`, "method", "inheritance", "constructor",…
- This results in a lot of confusion and misunderstanding.
- (Some might find learning javascript easier if they don't already know Java, C++, etc.)

javascript objects

- Object properties
- Inspecting object properties and values
- Prototype-based object "inheritance".
- Object prototypes.

Accessing Object Properties

```
let x = {name: 'Lou'} //literal object
console.log(x)  //{ name: 'Lou' }
x.name //'Lou'
x[name]  //ERROR
x['name']  //'Lou'
x["name"]  //'Lou'
```

accessing object properties

x.foo //here foo must be a javascript identifier (property access)
x[foo] //here foo must be a string (key access)

Objects and their properties

```
let x = {name: 'Lou'}
console.log(x)  //{ name: 'Lou' }
x.name = 'Louis' //re-assign
x.email = 'ldnel@scs.carleton.ca' //create
x['office'] = '5370 Herzberg' //create
console.log(x)
{ name: 'Louis',
  email: 'ldnel@scs.carleton.ca',
  office: '5370 Herzberg' }
```

Assigning and creating properties

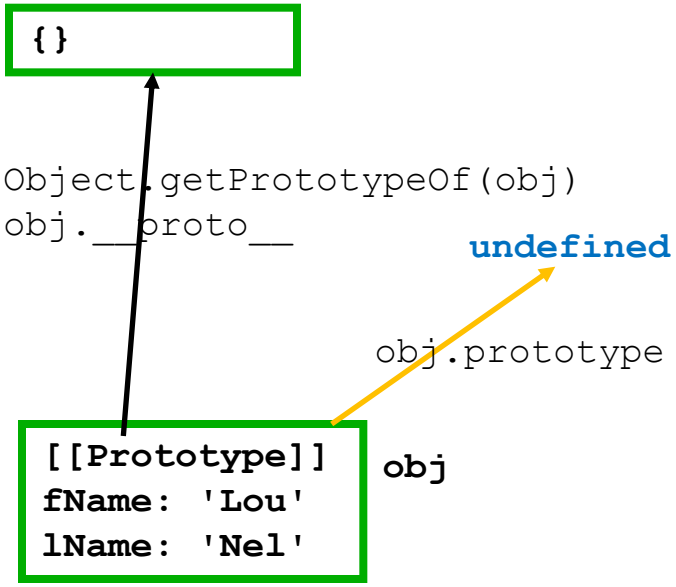
Objects and their Inheritance Prototypes

```
let p1 = {color: 'red'}
let x = {name: 'Lou'}
console.log(x)  //{ name: 'Lou' }
x.__proto__ = p1 //establish prototype link
//or
//Object.setPrototypeOf(x, p1); //newer style
//or
//var x = Object.create(p1); //newer style
console.log(x)  //{name: 'Lou'}
console.log(x.__proto__) //{color: 'red'}
console.log(Object.getPrototypeOf(x)) //{color: 'red'}
```

no rules relationship

But: what kind of relationship is x → p1 (it is an "is a" or a "part of" relationship?

Object Prototypes



Javascript has a hidden property within objects called **[[Prototype]]** to refer to the inheritance prototype of the object.

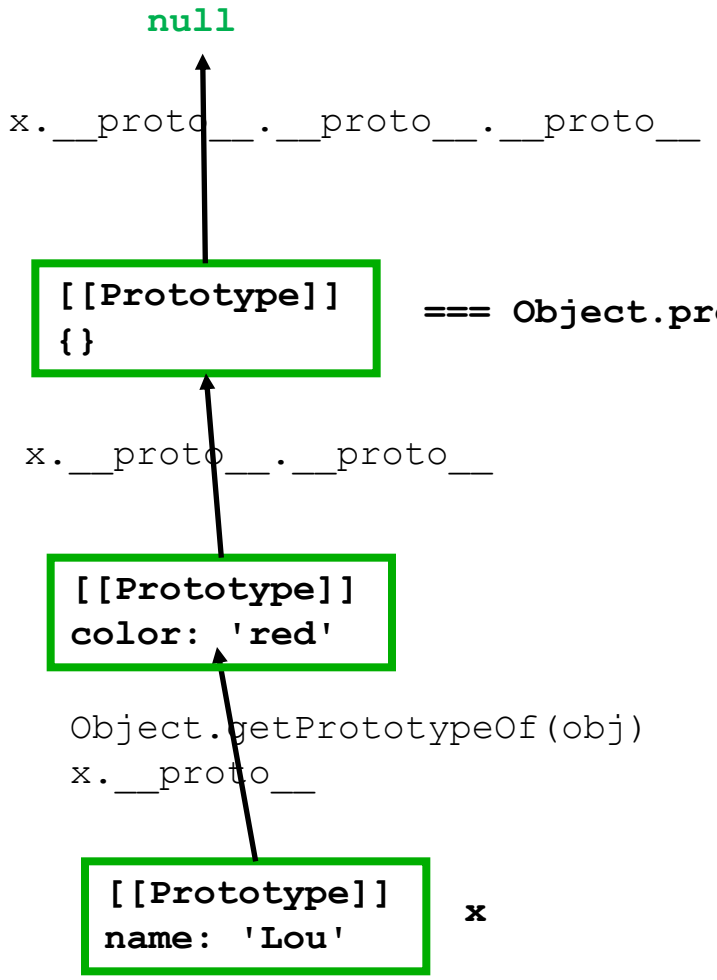
Two ways of accessing the prototype:
`obj.__proto__`
and
`Object.getPrototypeOf(obj)`

There is also an **`obj.prototype`** property but that only applies if obj is a function, otherwise `obj.prototype` is `undefined` (does not exist in non-function objects).

Very confusing:
`obj.prototype` is **not** the inheritance prototype of obj.

`.prototype` is probably a poorly chosen name for what `.prototype` actually represents. …more on this later.

Objects and their Inheritance Prototypes




```
x
{ name: 'Lou' }
x.__proto__
{ color: 'red' }
x.__proto__.__proto__
{}
x.__proto__.__proto__.__proto__
null
```

Objects and their Inheritance Prototypes

```
let p1 = {color: 'red'}
let x = {name: 'Lou'}
console.log(x);  //{ name: 'Lou' }
x.__proto__ = p1; //older style
//or
//Object.setPrototypeOf(x, p1); //newer style
//or
//var x = Object.create(p1); //newer style
console.log(x);  //{ name: 'Lou' }
console.log(x.__proto__); //{ color: 'red' }
console.log(x.name) //'Lou'
console.log(x.color) //'red'
```

Objects "inherit" the properties of their prototypes in that they are linked, by reference, to their prototypes.

Logging an object to the console does not reveal its inherited properties –important to remember since console.log() will be used often for debugging and inspection

Objects and their prototypes

```
let p1 = {color: 'red'}
let x = {name: 'Lou'}
console.log(x)  //{ name: 'Lou' }
x.__proto__ = p1
console.log(x)  //{ name: 'Lou' }
console.log(x.__proto__) //{ color: 'red' }

x.hasOwnProperty('name'); //true
x.hasOwnProperty('color'); //false

p1.isPrototypeOf(x); //true
```

Objects inherit the properties of their prototypes

`hasOwnProperty('key')` will not reveal inherited properties.
`p1.isPrototypeOf(x)` test whether p1 is somewhere on the prototype chain of x.

For-In loop will reveal inherited properties.

```
var p1 = {color: 'red'}
var x = {name: 'Lou'}
x.__proto__ = p1

for(k in x) console.log(k)
name
Color

for(k in x) console.log(`${k}: ${x[k]}`)
name: Lou
color: red
```

You can loop through the objects properties with a `for-in` loop and this **will** pick up the properties of the prototype.

This will be very useful to ensure you have examined the whole object: it's immediate properties and those inherited through prototypes.

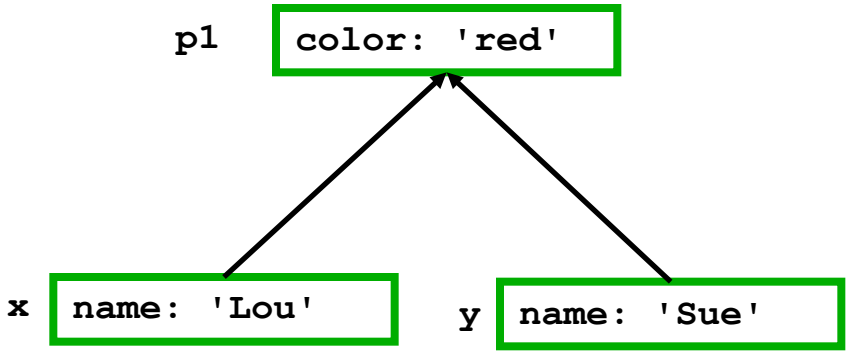
Objects and their prototypes

```
let p1 = {color: 'red'}
let x = {name: 'Lou'}
x.__proto__ = p1

for(k in x) console.log(k)
name
color
for(k in x) console.log(typeof k)
string
String

typeof x
'object'
```

The keys, or property names, are strings

Prototype Objects are Shared by Reference

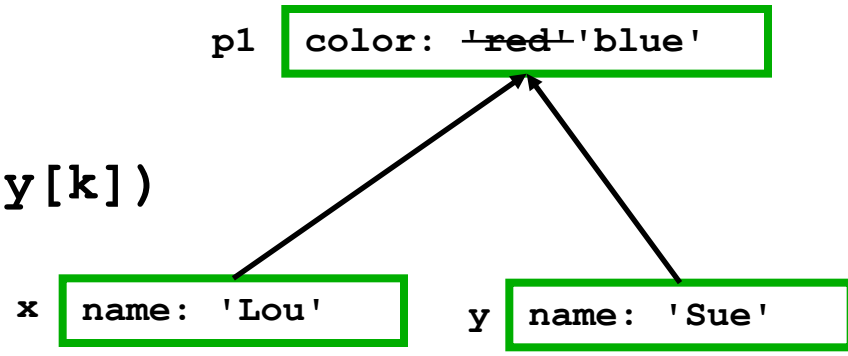
```
let p1 = {color: 'red'}
let x = {name: 'Lou'}
x.__proto__ = p1
let y = {name: 'Sue'}
y.__proto__ = p1
x //{ name: 'Lou' }
y //{ name: 'Sue' }
for(k in x) console.log(k + ": " + x[k])
name: Lou
color: red
for(k in y) console.log(k + ": " + y[k])
name: Sue
color: red
```



Prototypes are often shared by reference.
Helpful to simulate “classical” inheritance.

Modifying a Prototype

```
for(k in x) console.log(k+": "+ x[k])
name: Lou
color: red
for(k in y) console.log(k+": "+ y[k])
name: Sue
color: red
```

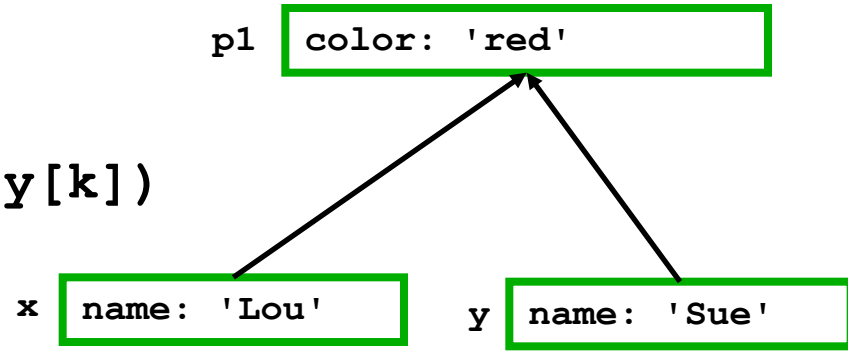



```
x.__proto__.color = 'blue'
for(k in x) console.log(k+": "+x[k])
name: Lou
color: blue
for(k in y) console.log(k+": "+y[k])
name: Sue
color: blue
```

Prototype shared by reference

What about?

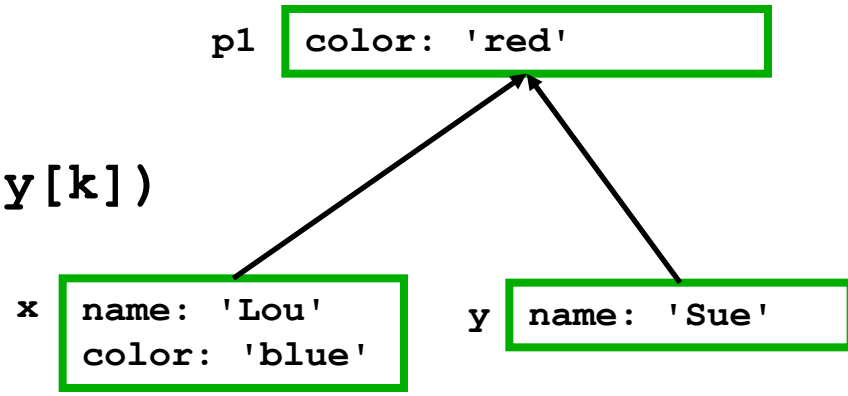
```
for(k in x) console.log(k+": "+ x[k])
name: Lou
color: red
for(k in y) console.log(k+": "+ y[k])
name: Sue
color: red

x.color = 'blue' //??
```



Property Shadowing

What about?

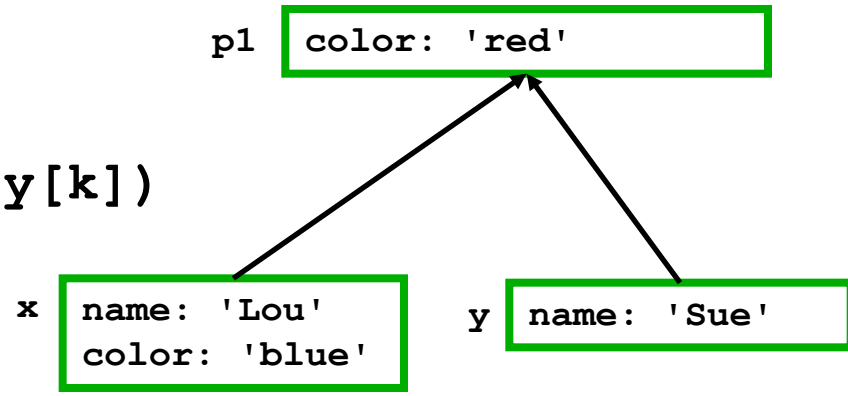
```
for(k in x) console.log(k+": "+ x[k])
name: Lou
color: red
for(k in y) console.log(k+": "+ y[k])
name: Sue
color: red
x.color = 'blue'
for(k in x) console.log(k+": "+x[k])
name: Lou
color: blue
for(k in y) console.log(k+": "+y[k])
name: Sue
color: red
```



Property Shadowing: x.color shadows p1.color

What about?

```
for(k in x) console.log(k+": "+ x[k])
name: Lou
color: red
for(k in y) console.log(k+": "+ y[k])
name: Sue
color: red
x.color = 'blue'

x.hasOwnProperty('color'); //true
y.hasOwnProperty('color'); //false
```



Property Shadowing: x.color shadows p1.color
Note this is the default behavior but can be overridden with custom getter/setter property functions (not discussed here).

Object-Based vs. Object-Oriented

Javascript is "object-based" as opposed to "object-oriented"

- **Objects do not derive from classes (types) but rather from reference links to other objects.**
- **Objects retain a connection to their ancestor prototypes which are themselves objects.**
- **Recall in javascript anything that is not: number, string, boolean, NaN or undefined is an object and always referred to by reference.**
- **Javascript has two special kinds of objects: function objects and arrays. (functions and arrays are objects in javascript and have some additional capabilities.)**
- **For example, function objects are callable, and array objects are indexable by integers, otherwise they are still just regular javascript objects.**

Object-Based

Javascript is “object-based” as opposed to “object-oriented”

- **Objects do not derive from classes (types) but rather from other objects.**

Consequences:

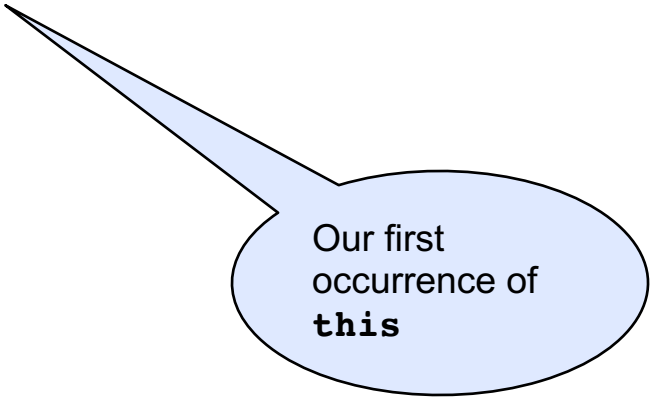
To add and remove properties of existing object you don't need to change, and recompile, a class definition.

You can add properties to existing objects and also remove them at any time.

You can add "methods" to existing objects or remove them. (Javascript does not really have methods –adding a function to an object does not make it a method, more on this later)

Properties can be Functions

```
var obj = {fName: 'Lou', lName: 'Nel'}
//add function
obj.getName = function(){return this.fName + " " + this.lName}

obj
{ fName: 'Lou',
  lName: 'Nel',
  getName: [Function] }

obj.getName //getName is a function
[Function]

obj.getName() //call getName function
'Lou Nel'
```



The properties of objects can be functions.
Functions are themselves also objects.

This leads to an variety arrangements of objects and functions

Object Function Properties added and removed

```
obj
{ fName: 'Lou', lName: 'Nel' }

//add a method to object
obj.getName = function(){return this.fName + " " + this.lName}

obj
{ fName: 'Lou',
  lName: 'Nel',
  getName: [Function] }

//call the "method"
obj.getName()  //'Lou Nel'

//remove the method
delete obj.getName
obj
{ fName: 'Lou', lName: 'Nel' }
```

Inspecting function properties

```
obj  //{fName: 'Lou', lName: 'Nel'}

//add a function to object
obj.getName = function(){return this.fName + " " + this.lName}

console.log(x.getName)
[Function]

console.log(x.getName.toString())
function (){return this.fName + " " + this.lName}
```

We can obtain a string representation of the function code.

Again, handy for inspection or debugging.

This is potentially a big security hole.

What's the meaning of `this`?

```
> var obj = {fName: 'Lou', lName: 'Nel'}
//add function
> obj.getName = function(){return this.fName + " " + this.lName;}

> obj;
{ fName: 'Lou',
  lName: 'Nel',
  getName: [Function] }

> obj.getName; //getName is a function
[Function]

> obj.getName(); //call getName function
'Lou Nel'
```

Be careful with **`this`**.
It is more complicated than in object-oriented languages like Java, C++, etc.
It would seem like getName() is a method of obj. but in javascript there are no methods –there is no special ownership that obj has over getName().

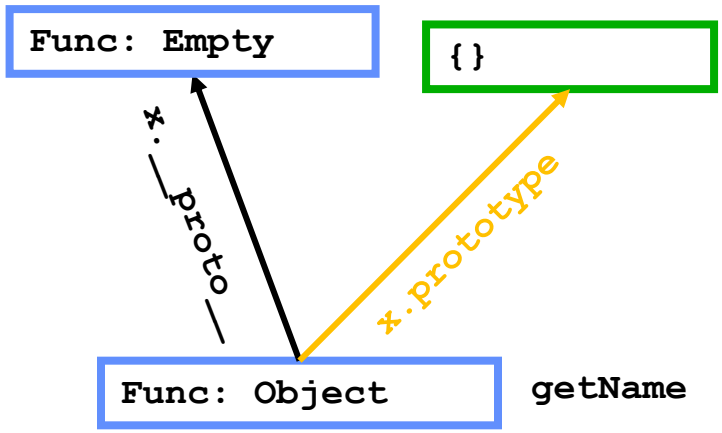
Object prototypes

Javascript appears to provide three mechanisms to obtain an object's prototype:

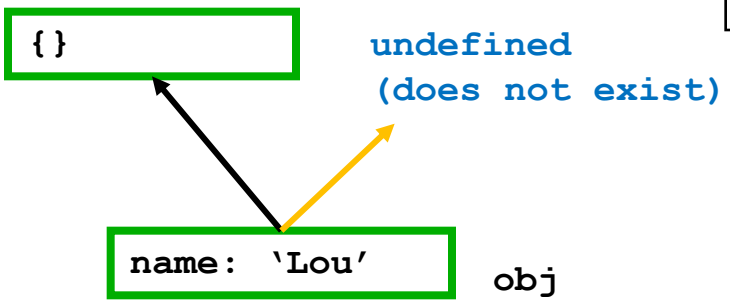
- `Object.getPrototypeOf(x)`
- `x.__proto__`
- `x.prototype`

- **We examine these as applied to both objects and function objects.**

Objects and Their Prototypes




```
let obj = {name: 'Lou'}
let getName = function(){return this.name}
obj  //{ name: 'Lou' }
obj.__proto__  //{}
obj.prototype //undefined

getName.__proto__ //[Function: Empty]
getName.prototype  //{}
```




```
Object.getPrototypeOf(obj)
obj.__proto__;
```

obj.prototype →

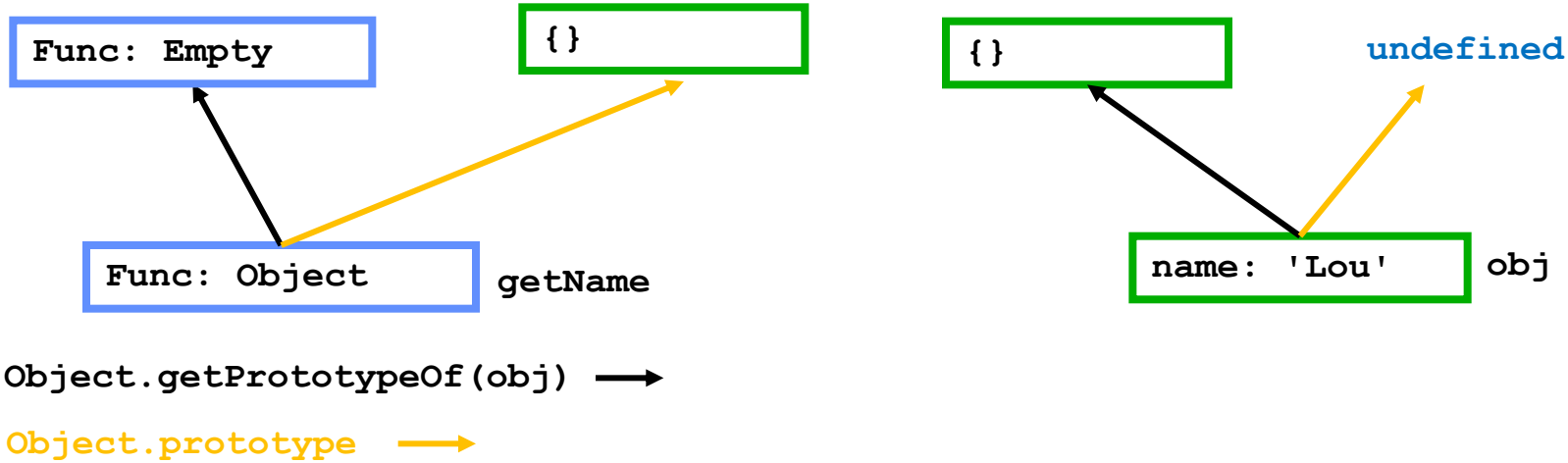
It would appear that functions have a `.prototype` property that non-function objects do not.

What would it be used for?

Objects vs Function Objects

Why do non-function objects have only one prototype while function objects have two?

Objects and Their Prototypes



Observations:

The **`func.prototype`** of function objects acts as prototype for the construction of non-function objects when using the function as a **constructor**. (Any function in javascript can be used as a constructor by preceeding the invocation with **`new`**.)

The **`Object.getPrototypeOf(func)`** of function object acts as inheritance prototype of the existing function treated as an object.

Objects vs Function Objects

Lets examine the fundamental differences between function objects and non-function objects

Objects vs Function Objects

```
let obj = {…}
```

What capabilities do all objects have?

- -data properties `obj.name = 'Lou'`
- -method properties `obj.getName = function(){return this.name}`
- -ancestor prototypes to inherit from

```
Object.getPrototypeOf(obj)
obj.__proto__
```

Objects vs Function Objects

What special capabilities do function objects have?

- **function objects are objects so they still have:**
  - **-data properties**
  - **-method properties**
  - **-ancestor prototypes to inherit from**

But Also:

- **functions can be invoked (called)**

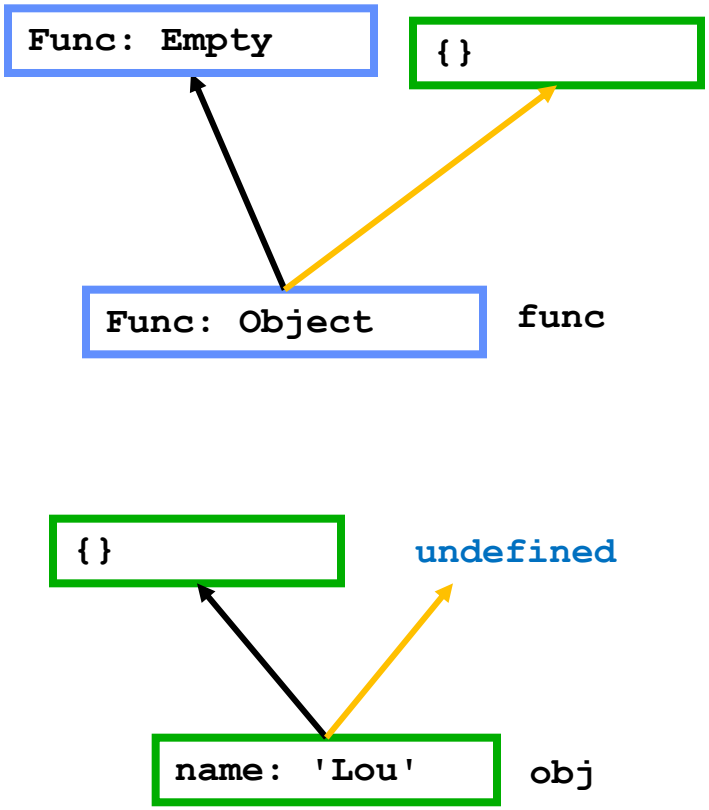
```
let x = func()
```

- **functions can instantiated: act as constructors (any function can be a constructor if the call is preceded with `new`.)**

```
let x = new func()
```

- **(ES6 => (arrow) functions cannot be used as constructors)**

Objects and Their Prototypes

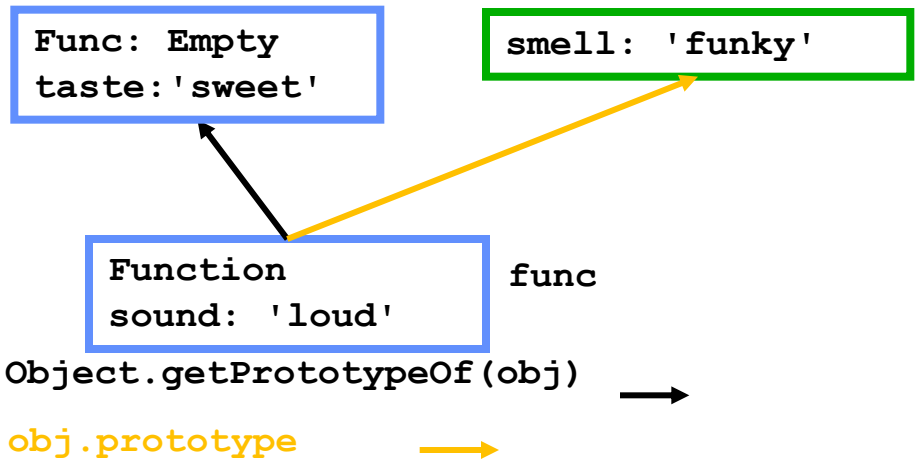


Function objects have one prototype to act as their ancestor when the being used as an object.

They have another prototype that serves a the prototype object for new objects created when they are acting as a constructor (being invoked with `new`).

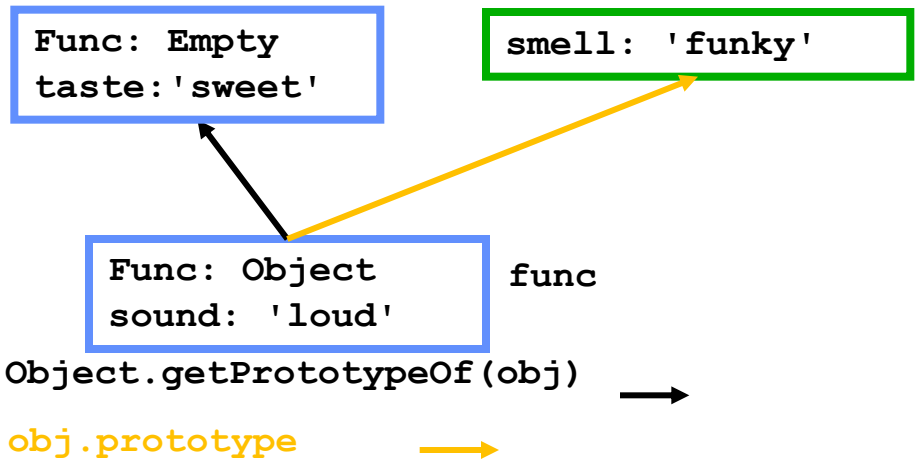
Neither prototype is necessarily required when they are simply being invoked (called)

Adding properties



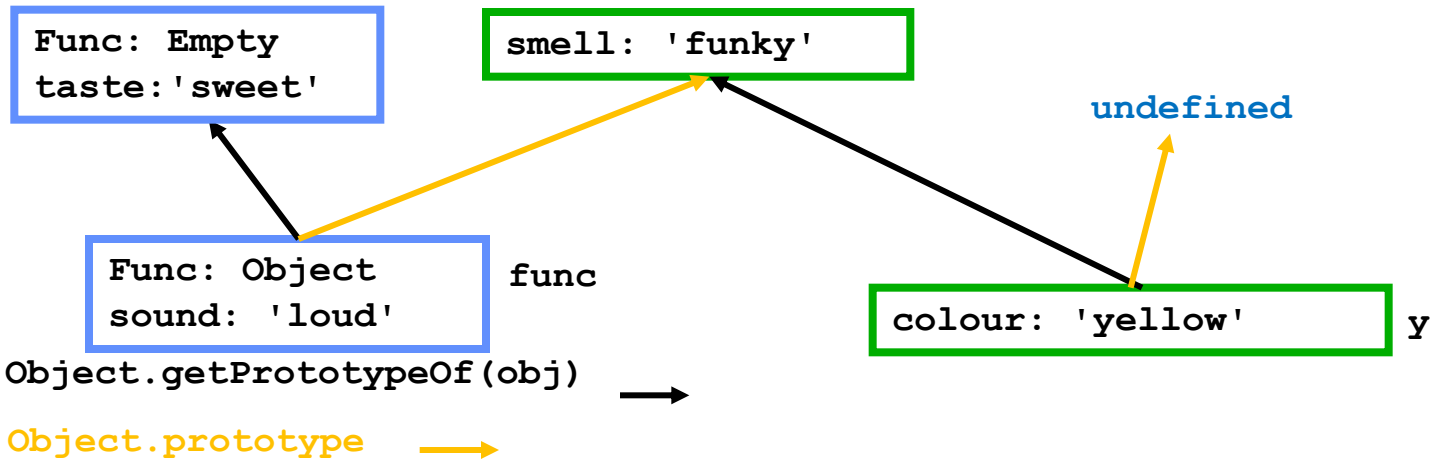

```
var func = function(x){this.colour = x}  //define function

func.sound = 'loud' //add object property to function
(func.__proto__).taste = 'sweet' //add prop to inheritance object
func.prototype.smell = 'funky' //add prop to construction prototype
```

Using Function as a Constructor



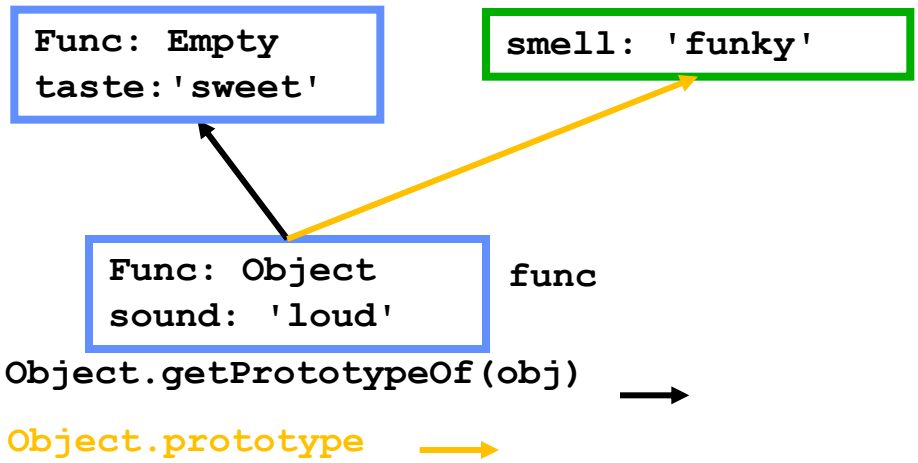

```
let func = function(x){this.colour = x}

func.sound = 'loud'
(func.__proto__).taste = 'sweet'
func.prototype.smell = 'funky'

var y = new func('yellow')  //<-- what does this produce?
```

Function as a Constructor



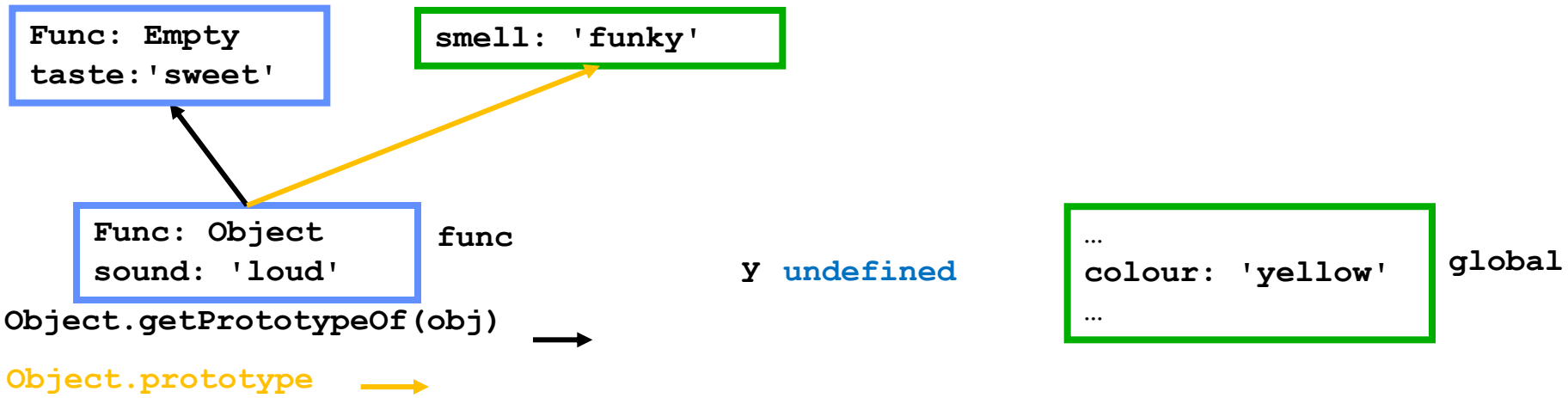

```
var func = function(x){this.colour = x}

(func.__proto__).taste = 'sweet'
func.prototype.smell = 'funky'
func.sound = 'loud'

var y = new func('yellow')  //<-- what does this produce?
```

What about Plain Invocation (no use of `new`)



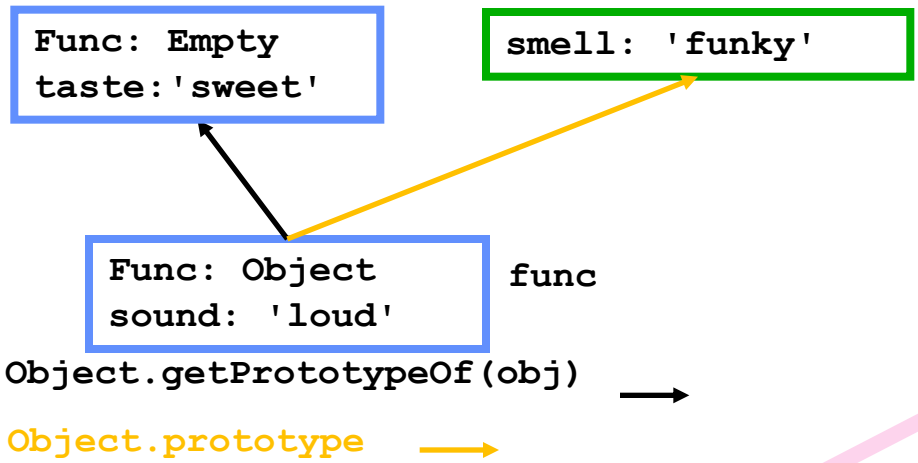

```
var func = function(x){this.colour = x}

func.sound = 'loud'
(func.__proto__).taste = 'sweet'
func.prototype.smell = 'funky'

var y = func('yellow')  //<-- what does this produce?
                        //notice no use of "new"
```

What about plain invocation




```
var obj = {size: "Big"}
var func = function(x){this.colour = x}

func.sound = 'loud'
(func.__proto__).taste = 'sweet'
func.prototype.smell = 'funky'

var y = func('yellow')  //<-- what does this produce? notice no"new"
console.log(y) //undefined
console.log(global.colour) //'yellow'
```

What about plain invocation



What does `this` refer to in this context?

Answer: the `global` object

```
var obj = {size: "Big"}
var func = function(x){this.colour = x}

func.sound = 'loud'
(func.__proto__).taste = 'sweet'
func.prototype.smell = 'funky'

var y = func('yellow')
```

Function Invocation

There are five basic ways to invoke a function in javascript:

As a "method"

As a function call

As a constructor

As an `apply`, or `call`

As => function (ES6 arrow function)

These methods differ in what `this` is considered to be.

Function Invocation

Five basic ways to invoke a function. These differ in what `this` turns out to be:

As a method: `obj.func();` **`//this` is `obj`**

As a function call: `func();` **`//this` is the `gobal` object –ugly!!**

As a constructor: `new func();` **`//this` is the new object created**

As an apply, or call:

```
func.apply(anObj, arg1);              //this is anObj
func.call(anObj, arg1, arg2, …);      //this is anObj
```

As an arrow function:

`(k,v)=>{this[k]=v}` **`//this` is `that` (borrowed from enclosing scope)**

```
let x = {name:'Lou'}
let f = function(k,v){this[k]=v}
x.method = f

x.method('color','red')
x   //{ name: 'Lou', method: [Function: f], color: 'red' }

f('name','Sue')
global.name   //'Sue'

let y = new f('size','big')
y   //f { size: 'big' }

let z = new (x.method)('size','small')
z    //f { size: 'small' }
x    //{ name: 'Lou', method: [Function: f], color: 'red' }

let w = {}
x.method.call(w,'condition','used')
w  // { condition: 'used' }
x  // { name: 'Lou', method: [Function: f], color: 'red' }
```

This and That

```
function Car(){
   this.set = function(k,v){this[k]=v}
}

let c = new Car()
c   //Car { set: [Function] }
c.set('colour','black')
c   //Car { set: [Function], colour: 'black' }
let f = c.set
f('engine','v8')
c  //Car { set: [Function], colour: 'black' }
   //what did the invocation affect?
let x = new f('size','large')
x   //{size: 'large'}
c   //{ set: [Function], colour: 'black'}
```

This example shows that a “method” of an object can be used to affect another `this` object

This and That

```
function Truck(){
    let that = this
    this.set=function(k,v){
       that[k]=v
    }
}

let t = new Truck()
t    //Truck { set: [Function] }
let f= t.set
f('size','big')
t   //Truck { set: [Function], size: 'big' }
let x = new f('engine','v8')
x  //{}
t  //Truck { set:[Function], colour:'red', size:'big',
engine:'v8'}
```

This shows the famous `that=this` trick that is used to ensure that the method always accesses the `this` of object it was allocated to.

i.e. `that` forms a closure around `this`.

This and That

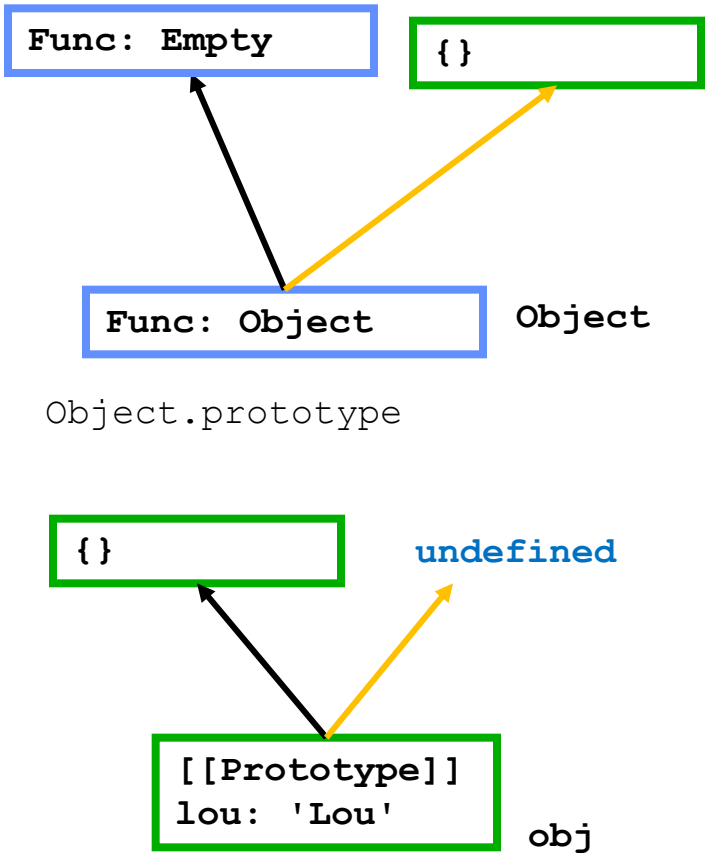
```
function Car(){
   this.set = (k,v)=>{this[k]=v}
}

let c = new Car()
c   //Car { set: [Function] }
c.set('colour','black')
c   //Car { set: [Function], colour: 'black' }
let f = c.set
f('engine','v8')
c  //Car { set: [Function], colour: 'black', engine: 'v8' }

let x = new f('size','large') //RUNTIME ERROR
```

This example shows that ES6 arrow function methods implicitily map `this` to `that`

More specifically, => function `this` binds to enclosing function's `this`

Moreover => functions cannot be used a constructors

Objects and Their Prototypes -Summary



Functions are objects:

Function objects have one prototype to act as their ancestor when the being used as an object.

Functions can be Constructors:

They have another prototype that serves a the prototype object for new objects created when used as a constructor (being invoked with `new`).

Functions can be Invoked:

Neither prototype is necessarily required when they are simply being invoked.(called)

There are 5 invocation patterns which determine what is used for `this`.

Pseudoclassical Construction

- Function objects intended to act as constructors are often defined as follows: (Crockford calls this "pseudoclassical construction" –it's sort of like classes in Java, C++)
- When javacript people talk about classes this is what they mean.

```
var Car = function(v,m,c){
  this.vin = v
  this.make = m
  this.colour = c
}

Car.prototype.getColour = function(return this.colour)
Car.prototype.getModel = function(){return this.make}


//Later….
var myCar = new Car(1000, 'Toyota','Black')
```

Pseudoclassical Construction

```
var Car = function(v,m,c){
  this.vin = v
  this.make = m
  this.colour = c
}

Car.prototype.getColour = function(return this.colour)
Car.prototype.getModel = function(){return this.make}


//Later….
var myCar = new Car(1000, 'Toyota','Black')
```

It is tradition for function objects that are meant to be constructors to be named with identifiers that start with a capital letter.

Makes them more reminiscent of classes in Java or C++ -- I guess.

Makes programmers aware they are meant to used with `new`.

Javascript has no idea whether you intend functions to act as constructors or not (all functions can be used as constructors)

ES6 Classes

Javascript ES6 has recently introduced the concept of a class

```
class Set{
constructor(){
      this.collection = []
   }
   add(x){ //add element x if no current element === x
      if(this.collection.indexOf(x) < 0) this.collection.push(x)
      }
   remove(x) { //remove first occurence of element === x
      var position = this.collection.indexOf(x)
      if(position > -1) this.collection.splice(position,1)
      }
   contains(x){ //answer whether set contains element === x
      return this.collection.indexOf(x) > -1
      }
}
```

ES6 classes. This is "syntactic sugar" for classes built from functions and does **not** solve the encapsulation issues -more on them later.

.prototype and .constructor

```
function Foo(){/*stuff*/}

var obj = new Foo()

obj.__proto__ === Foo.prototype; //true
Object.getPrototypeOf(obj) === Foo.prototype; //true
```

So `Foo.prototype` is the object that will serve as `obj`'s inheritance prototype. (Surely a better name could have been chosen for this!)

```
obj.constructor === Foo //true
Foo.prototype.constructor === Foo //true
```

So `obj.constructor` is the function object that was used with `new` to create `obj`.

(`.constructor` is a hidden, non-enumerateable property)