

# JavaScript Training

Module I - Core Concepts

# Overview

- Object
- Functions
- Inheritance

# Useful stuff

- Console API
  - `console.log(object[, object, ...])`  
Writes a message to the console
  - `console.dir(object)`  
Prints an interactive listing of all properties of the object
- `debugger`  
Adds a breakpoint
- Use `===` and `!==` Never use `==` and `!=`

# Object Literal

```
var empty_object = {};  
  
var person = {  
    "first-name": "Jerome",  
    "last-name": "Howard",  
    "prefix": "Mr"  
};
```

# Retrieval

```
person["first-name"]    // "Jerome"
```

```
person["prefix"]        // "Mr"
```

```
person.prefix           // "Mr"
```

```
// Default values
```

```
var middle = person["middle-name"] || "(none)";
```

```
// Guard against undefined values
```

```
person.father            // undefined
```

```
person.father.name       // throw "TypeError"
```

```
person.father && person.father.name // undefined
```

# Update

- A value in an object can be updated by assignment.

```
person['first-name'] = 'Jerome';
```

- If the object does not already have that property name, the object is augmented:

```
person['middle-name'] = 'Lester';
```

```
person.nickname = 'Curly';
```

```
flight.equipment = {  
    model: 'Boeing 777'
```

```
};
```

```
flight.status = 'overdue';
```

# Reference

Objects are passed around by reference. They are never copied:

```
var x = person;

x.nickname = 'Curly';

var nick = person.nickname;

    // nick is 'Curly' because x and person
    // are references to the same object

var a = {}, b = {}, c = {};

    // a, b, and c each refer to a
    // different empty object

a = b = c = {};

    // a, b, and c all refer to
    // the same empty object
```

# Prototype

- Every object is linked to a prototype object from which it can inherit properties.
- All objects created from object literals are linked to `Object.prototype`, an object that comes standard with JavaScript.
- If we try to retrieve a property value from an object, and if the object lacks the property name, then JavaScript attempts to retrieve the property value from the prototype object.



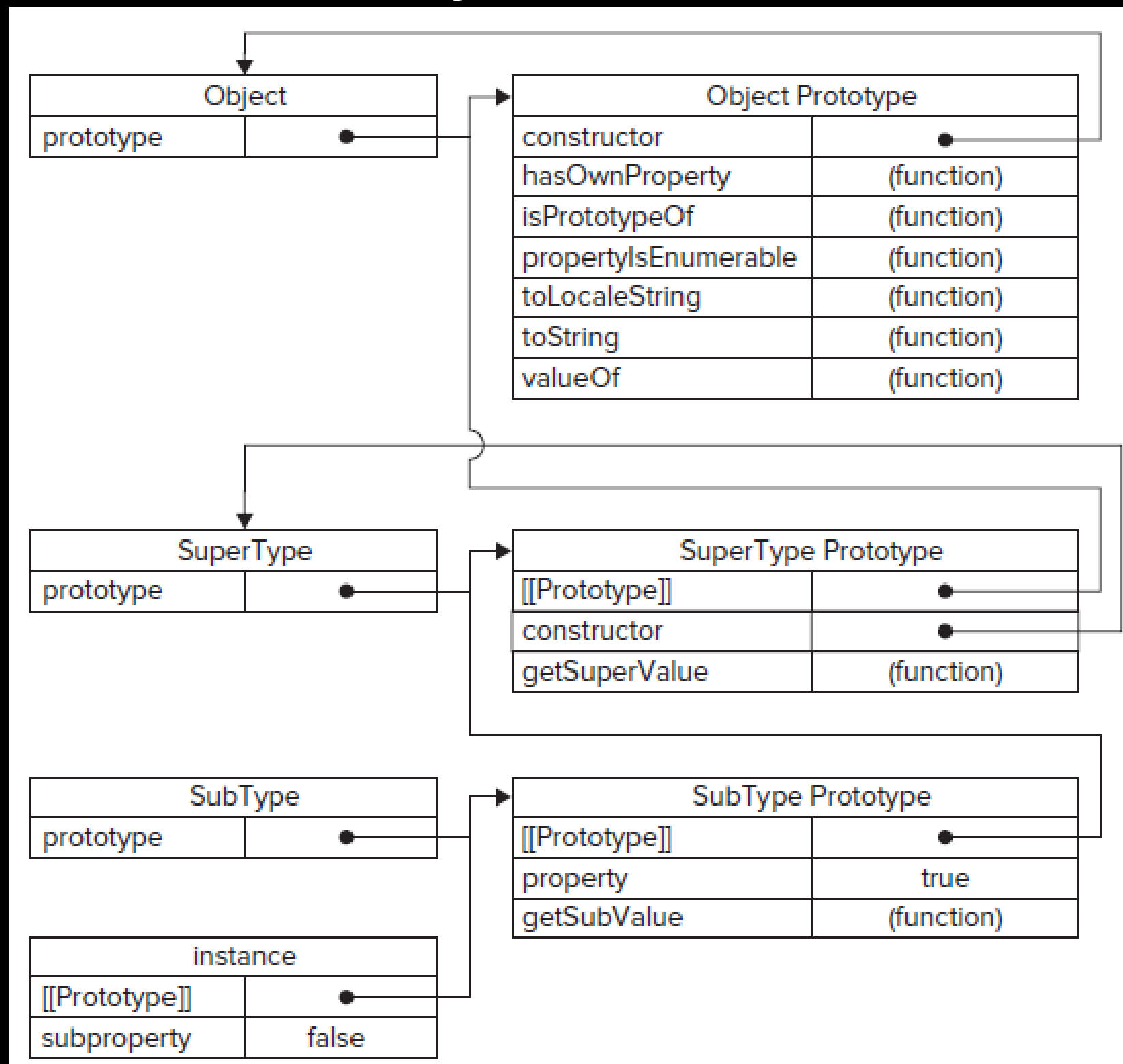
# Prototype

```
if (typeof Object.beget !== 'function') {  
    // prevent function re-definition  
    Object.beget = function (o) {  
        var F = function () {};  
        F.prototype = o;  
        return new F();  
    };  
}
```

```
var another_person = Object.beget(person);
```

```
another_person['first-name'] = 'Harry';  
another_person['middle-name'] = 'Moses';  
another_person.nickname = 'Moe';
```

# Prototype Chain



# Prototype Chain

```
> var Person = function(name) {  
    this.name = name;  
};  
  
p = new Person('nombre');  
▶ Person  
  
> console.dir(p)  
▼ Person  
  name: "nombre"  
  ▼ __proto__: Object  
    ▶ constructor: function (name) {  
      ▼ __proto__: Object  
        ▶ __defineGetter__: function __defineGetter__() { [native code] }  
        ▶ __defineSetter__: function __defineSetter__() { [native code] }  
        ▶ __lookupGetter__: function __lookupGetter__() { [native code] }  
        ▶ __lookupSetter__: function __lookupSetter__() { [native code] }  
        ▶ constructor: function Object() { [native code] }  
        ▶ hasOwnProperty: function hasOwnProperty() { [native code] }  
        ▶ isPrototypeOf: function isPrototypeOf() { [native code] }  
        ▶ propertyIsEnumerable: function propertyIsEnumerable() { [native code] }  
        ▶ toLocaleString: function toLocaleString() { [native code] }  
        ▶ toString: function toString() { [native code] }  
        ▶ valueOf: function valueOf() { [native code] }
```

# Pseudo-Class

A JavaScript Pseudo Class can be created by declaring a function and calling that function with the new operator. The new operator returns a copy of the function's prototype

```
function MyType(val) {  
    this.attr = val;  
}
```

```
MyType.prototype.method = function() {  
    return "called method()";  
}
```

```
var typeInstance = new MyType("value");  
  
console.log(typeInstance.method());
```

# Reflection

The `typeof` operator can be very helpful in determining the type of a property:

```
typeof person.nickname      // 'string'
```

Some care must be taken because any property on the prototype chain can produce a value:

```
typeof person.toString      // 'function'  
typeof person.constructor   // 'function'
```

The other approach is to use the `hasOwnProperty` method, which returns `true` if the object has a particular property. The `hasOwnProperty` method does not look at the prototype chain:

```
person.hasOwnProperty('nickname')      // true  
person.hasOwnProperty('constructor')   // false
```

# Enumeration

The for in statement can loop over all of the property names in an object.

```
var name;
for (name in person) {
    // We can show only this object properties
    // using hasOwnProperty
    if (typeof person[name] !== 'function') {
        console.log(name + ': ' +
person[name]);
    }
}
```

# Delete

- The delete operator can be used to remove a property from an object. It will remove a property from the object if it has one. It will not touch any of the objects in the prototype linkage.
- Removing a property from an object may allow a property from the prototype linkage to shine through:

```
another_person.nickname    // 'Moe'
```

```
// Remove nickname from another_person, revealing  
// the nickname of the prototype.
```

```
delete another_person.nickname;
```

```
another_person.nickname    // 'Curly'
```

# Global variables

One way to minimize the use of global variables is to create a single global variable for your application:

```
var MYAPP = {};  
  
MYAPP.person = {  
  "first-name": "Joe",  
  "last-name": "Howard"  
};  
  
MYAPP.flight = {  
  airline: "Oceanic",  
  departure: {  
    IATA: "SYD",  
    time: "2004-09-22 14:55",  
    city: "Sydney"  
  },  
};
```



# Exercise - Prototype chain

```
var Person = function(name) {  
  this.name = name;  
};  
  
p = new Person('nombre');  
console.log(p.something); // ???  
Person.prototype.something = 'algo'  
console.log(p.something); // ???
```

---

```
var Person = function(name) {  
  this.name = name;  
};  
  
var p = new Person('my name');  
p.something = 'oooo';  
console.log(p.something); // ???  
Person.prototype.something = 'my something';  
console.log(p.something); // ???  
  
var p2 = new Person();  
console.log(p2.something); // ???
```

# Exercise - Extend

Copy all of the properties in the source objects over to the destination object, and return the destination object.

```
utils.extend({name : 'moe'}, {age :  
50});
```

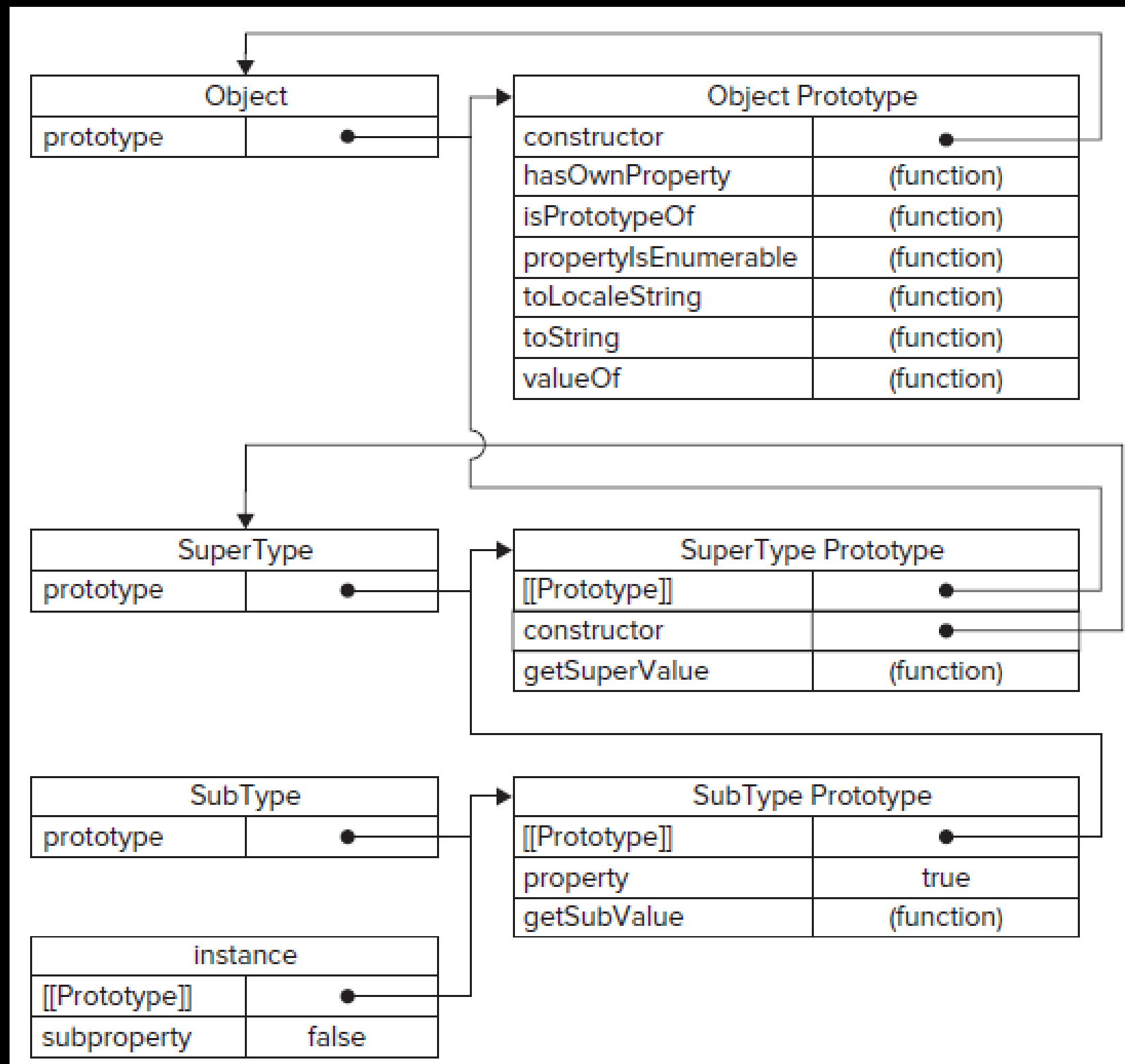
```
=> {name : 'moe', age : 50}
```

# Functions

# Function Objects

- Functions are objects that in addition can be invoked
- Function objects are linked to the Function.prototype (which is linked to the Object.prototype)
- Additional hidden properties
  - the function context
  - the code that implements the function
- constructor is a Function.prototype property whose value is the function (see Inheritance)

# Prototype Chain



# Function Literal

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

# Invocation

- The Method Invocation
- The Function Invocation
- The Constructor Invocation
- The Apply Invocation

# Method Invocation

When a function is stored as a property of an object, we call it a method.  
When a method is invoked, this is bound to that object.

```
var myObject = {  
  value: 0,  
  increment: function (inc) {  
    this.value += typeof inc === 'number' ? inc : 1;  
  }  
};
```

```
myObject.increment( );  
console.log(myObject.value);    // 1
```

```
myObject.increment(2);  
console.log(myObject.value);    // 3
```



# Function Invocation

When a function is not the property of an object, then it is invoked as a function:

```
var sum = add(3, 4);    // sum is 7
```

When a function is invoked with this pattern, *this* is bound to the global object.

When an inner function is invoked, *this* is bound to the global object

# Function Invocation

```
// Augment myObject with a double method.

myObject.double = function ( ) {
    var that = this;    // Workaround.
    var helper = function ( ) { // Inner function
        // try to do console.log(this);
        // to check this value
        that.value = add(that.value, that.value)
    };
    helper( );    // Invoke helper as a function.
};

// Invoke double as a method.

myObject.double( );

console.log(myObject.value);    // 6
```

# Constructor Invocation

- If a function is invoked with the *new* prefix, then a new object will be created with a hidden link to the value of the function's prototype member, and *this* will be bound to that new object.

# Constructor Invocation

```
var Quo = function (string) {  
    this.status = string;  
};  
  
// Give all instances of Quo a public method  
// called get_status.  
  
Quo.prototype.getStatus = function ( ) {  
    return this.status;  
};  
  
// Make an instance of Quo.  
  
var myQuo = new Quo("confused");  
console.log(myQuo.getStatus( )); // confused
```

# Apply Invocation

- The apply method lets us construct an array of arguments to use to invoke a function. It also lets us choose the value of *this*. The apply method takes two parameters. The first is the value that should be bound to this. The second is an array of parameters.

# Apply Invocation

```
// Make an array of 2 numbers and add them.  
var array = [3, 4];  
var sum = add.apply(null, array);    // sum is 7  
  
// Make an object with a status member.  
var statusObject = {  
    status: 'A-OK'  
};  
  
// statusObject does not inherit from Quo.prototype,  
// but we can invoke the get_status method on  
// statusObject even though statusObject does not have  
// a get_status method.  
  
var status = Quo.prototype.get_status.apply(statusObject);  
// status is 'A-OK'
```

# The arguments parameter

arguments is an array-like object without the array methods

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

```
console.log(sum(4, 8, 15, 16, 23, 42)); // 108
```

# Exceptions

```
var add = function (a, b) {  
  if (typeof a !== 'number' || typeof b !== 'number') {  
    throw {  
      name: 'TypeError',  
      message: 'add needs numbers'  
    } // also see new Error(message)  
  }  
  return a + b;  
}  
  
var try_it = function ( ) {  
  try {  
    add("seven");  
  } catch (e) {  
    console.log(e.name + ': ' + e.message);  
  }  
}  
  
tryIt( );
```



# Augmenting Types

JS allows basic types (functions, array, strings, numbers, etc) to be augmented

```
if (!String.prototype.myTrim) {  
    String.prototype.myTrim = function ( ) {  
        return this.replace(/^\s+|\s+$/g,  
        '' );  
    };  
}
```

```
console.log("    something    ");
```

```
console.log("    something    ".myTrim( ));
```

# Scope

- JS does have function scope
- That means that
  - The parameters and variables defined in a function are not visible outside of the function
  - And, that a variable defined anywhere within a function is visible everywhere within the function.
- Therefore, it is best to declare all of the variables used in a function at the top of the function body
- **Exception** with *this* and *arguments*

# Scope

```
var foo = function ( ) {  
  
    var a = 3, b = 5;  
    var bar = function ( ) {  
        var b = 7, c = 11;  
        // At this point, a is 3, b is 7, and c is 11  
  
        a += b + c;  
        // At this point, a is 21, b is 7, and c is 11  
    };  
  
    // At this point, a is 3, b is 5, and c is not defined  
    bar( );  
  
    // At this point, a is 21, b is 5  
  
};
```

# Closure

Closures are functions that have access to variables from another function's scope whose execution has finished.

```
function sayHello(name) {  
    var text = 'Hello ' + name; // local variable  
    var sayAlert = function() { console.log(text); }  
    return sayAlert;  
}  
  
var saySomething = sayHello('some name');  
  
// sayHello execution finished  
  
saySomething(); // 'Hello some name'
```

# Exercise - Closure

```
// Html
...
<p>1</p><p>2</p><p>3</p><p>4</p><p>5</p>
...

// What's the value of console.log(i)
// when I click on <p>5</p>???
// If the value is not the expected, How would you refactor the code
// so when you click on each p the console displays the p index?

var addHandlers = function(nodes) {
    var i;
    for ( i = 0; i < nodes.length; i += 1) {
        nodes[i].onclick = function(e) {
            console.log(i);
        }
    }
};

addHandlers(document.getElementsByTagName("p"));
```

# Callbacks

```
request = prepare_the_request( );  
// Slow server or network will leave the client  
// in a frozen state
```

```
response = send_request_synchronously(request);  
display(response);
```

```
// vs
```

```
request = prepare_the_request( );  
  
send_request_asynchronously(request, function (response) {  
    display(response);  
});
```

# Ajax

XMLHttpRequest (XHR) is an API available in web browser scripting languages such as JavaScript. It is used to send HTTP or HTTPS requests directly to a web server and load the server response data directly back into the script.

```
xhr = new XMLHttpRequest();

xhr.onreadystatechange = function() {
    console.log("XHR.readyState: " + xhr.readyState);

    if (xhr.readyState === 4) { // DONE
        var user = JSON.parse(xhr.responseText);
        document.getElementById("firstName").innerText = user["first-name"];
        document.getElementById("coreId").innerText = user.coreId;
    }
};

xhr.open("GET", "ajax.json", true); // true means async

xhr.send(null);
```

# Exercise - Async and Sync

- Given an existing API composed by 3 methods (m1, m2, m3), it is required to let the consumer of that api to be able to consume that API synchronously or asynchronously
- How will you design the solution for that problem?



# Question

```
var x = function () {  
    return 2;  
};
```

```
var y = function () {  
    return 2;  
}();
```

// What is the value of x and y?

```
console.log(x);
```

```
console.log(y);
```

# Module

A module is a function or object that presents an interface but that hides its state and implementation

```
var myModule = function () {  
    var privateVar01 = 'A', // private variables  
        privateVar02 = 'B',  
        privateFunction01 = function() {}, // private  
functions  
        privateFunction01 = function() {};  
  
    return { // public interface  
        publicFunction01: function ( ) {  
            },  
        publicFunction02: function ( ) {  
            },  
    };  
} ( );
```

# Module

```
var serialMaker = function ( ) {  
  var prefix = '';  
  var seq = 0;  
  return {  
    setPrefix: function (p) {  
      prefix = String(p);  
    },  
    setSeq: function (s) {  
      seq = s;  
    },  
    gensym: function ( ) {  
      var result = prefix + seq;  
      seq += 1;  
      return result;  
    }  
  };  
} ( );
```

# Module

```
serialMaker.setPrefix('Q');
```

```
serialMaker.setSeq(1000);
```

```
console.log(serialMaker.gensym( ));  
// Q1000
```

```
// How can I access prefix or seq from  
outside?
```

# Cascade (or Chaining)

Return this on those methods that set or change the state of an object allowing to call these methods in a single statement

```
var serialMaker = function ( ) {  
    // ...  
    return {  
        setPrefix: function (p) {  
            prefix = String(p);  
            return this;  
        },  
        setSeq: function (s) {  
            seq = s;  
            return this;  
        },  
        // ...  
    };  
} ( );
```

# Cascade (or Chaining)

```
serialMaker  
  
    .setPrefix('Q')  
  
    .setSeq(1000);  
  
console.log(serialMaker.gensym( ));  
// Q1000
```

# Hoisting

The statement:

```
function foo( ) {}
```

means about the same thing as:

```
var foo = function foo( ) {};
```

The second form makes it clear that foo is a variable containing a function object. To use the language well, it is important to understand that functions are objects.

function statements are subject to hoisting. This means that regardless of where a function is placed, it is moved to the top of the scope in which it is defined. This relaxes the requirement that functions should be declared before used, which leads to sloppiness.

# Exercise - Hoisting

```
hola(); // ???  
var hola = function() {  
    console.log('hola 2');  
};  
  
function hola() {  
    console.log('hola');  
};  
  
=====
```

```
var hola = function() {  
    console.log('hola 2');  
};  
hola(); // ???  
  
function hola() {  
    console.log('hola');  
};
```

```
var hola = function() {  
    console.log('hola 2');  
};  
  
function hola() {  
    console.log('hola');  
};  
  
hola(); // ???
```



# Exercise - Bind

Bind a function to an object, meaning that whenever the function is called, the value of `this` will be the object.

```
var func = function(greeting) {  
    return greeting + ': ' + this.name;  
};  
  
func = utils.bind(func, {name : 'Matias'},  
    'hi');  
  
// Expected behavior  
func();  
=> 'hi: Matias'
```

# Exercise - Memoize

- Memoize a given function by caching the computed result.

- ```
var fibonacci = function(n) {  
  console.log('Fib ' + n);  
  return n < 2 ? n : fibonacci(n - 1) + fibonacci(n - 2);  
}
```

- ```
var memoizedFibonacci = utils.memoize(fibonacci);
```

- ```
// Expected behavior  
memoizedFibonacci(5);  
  
// 1st time, I should see "Fib 5",..., "Fib 0"  
memoizedFibonacci(5);  
  
// 2nd time, I shouldn't see any "Fib x"
```

# JavaScript Slow sync operations

- JSON.parse
  - localStorage I/O
- XMLHttpRequest\*
- DOM access
  - document.\*
  - element.\*

# setTimeout

- `var myVar;`
- `function myFunction(){`
- `myVar = setTimeout(function(){`
- `alert("Hello");`
- `},3000);`
- `}`
- `function myStopFunction(){`
- `clearTimeout(myVar);`
- `}`

# Exercise - Lazy function

Create and return a new lazy version of the passed function that will postpone its execution until after wait milliseconds have elapsed since the last time it was invoked.

```
// e.g. writing to localStorage is a sync operation
// Provided a single JS thread, many writes in short time
may block the UI
var storeToDb = function(jsonDb) {
    localStorage.setItem("db", JSON.stringify(jsonDb));
}

var lazyStoreToDb = utils.lazy(storeToDb, 1000);
// Expected behavior
lazyStoreToDb(someJson);
// passed 500ms and no write
lazyStoreToDb(someOtherJson);
// passed 1001ms and write
```

# Exercise - Lazy function

```
// Exercise context

var store = function(jsonDb) {

    localStorage.setItem("db", JSON.stringify(jsonDb));

}

var DB = {};

for (var i = 0; i < 1000; i++) {

    item = {

        id : i,

        name : "Item " + i

    };

    DB[item.name] = item;

    store(DB); // Constraint: you cannot move this call out of the for loop

}
```

# Inheritance

# Introduction

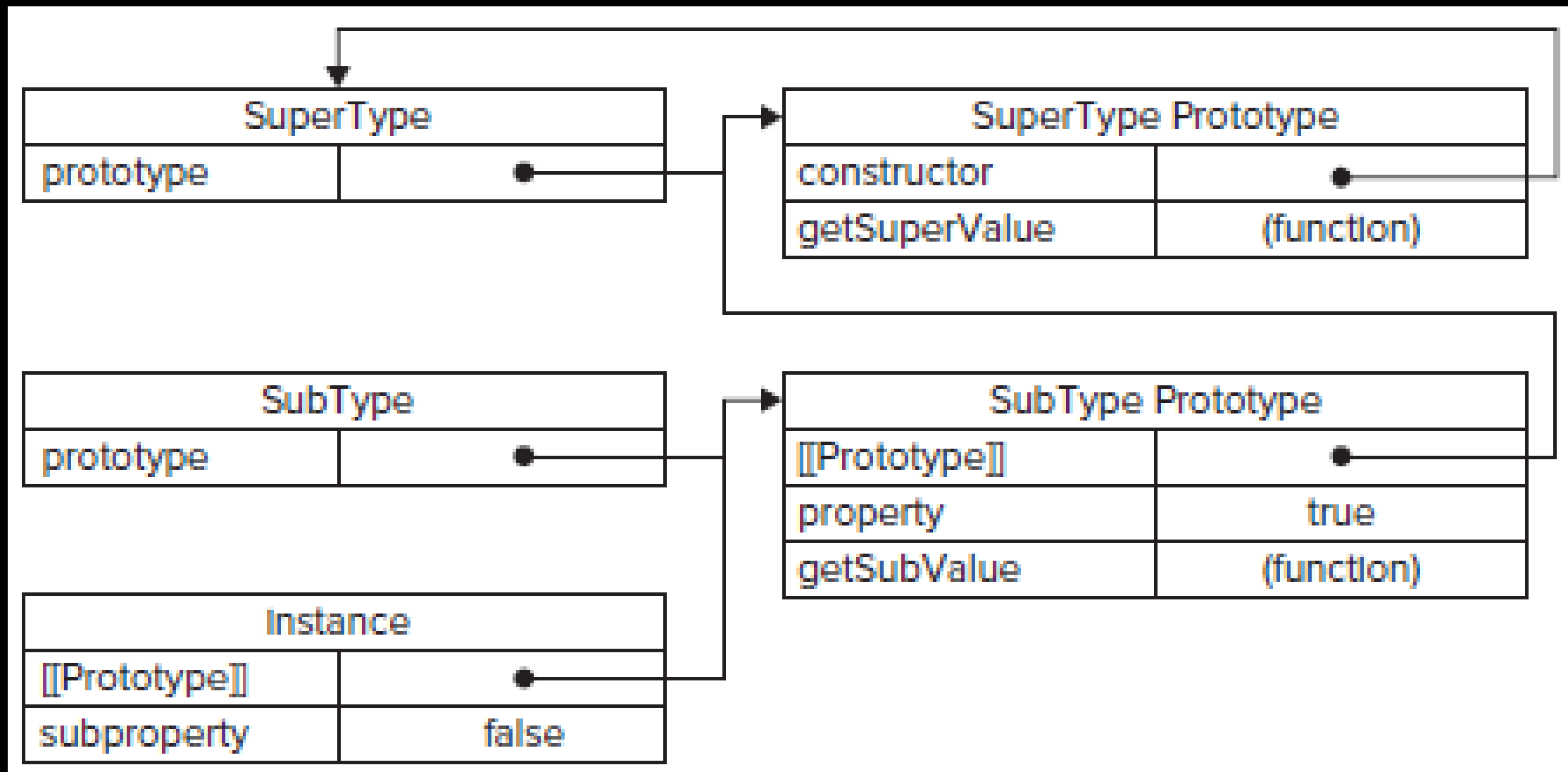
- "JavaScript, being a loosely typed language, never casts. The lineage of an object is irrelevant. What matters about an object is what it can do, not what it is descended from."
- "In classical languages, objects are instances of classes, and a class can inherit from another class. JavaScript is a prototypal language, which means that objects inherit directly from other objects."



# Prototype Chaining

```
function SuperType() {  
    this.property = true;  
}  
  
SuperType.prototype.getSuperValue = function() {  
    return this.property;  
};  
  
function SubType() {  
    this.subproperty = false;  
}  
  
//inherit from SuperType  
SubType.prototype = new SuperType();  
  
//new method  
SubType.prototype.getSubValue = function () {  
    return this.subproperty;  
};  
  
var instance = new SubType();  
alert(instance.getSuperValue()); //true
```

# Prototype Chaining



# Prototype Chaining - Issues

The major issue revolves around prototypes that contain reference values. The second issue is that the SubType cannot pass arguments to the SuperType constructor

```
function SuperType() {  
    this.colors = ["red", "blue", "green"];  
}  
  
function SubType() {}  
  
//inherit from SuperType  
SubType.prototype = new SuperType();  
  
var instance1 = new SubType();  
instance1.colors.push("black");  
  
alert(instance1.colors); // "red,blue,green,black"  
  
var instance2 = new SubType();  
alert(instance2.colors); // "red,blue,green,black"
```

# Constructor Stealing

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

```
function SubType() {  
    //inherit from SuperType passing in an argument  
    SuperType.call(this, "Nicholas");  
    //instance property  
    this.age = 29;  
}
```

```
var instance = new SubType();  
alert(instance.name); // "Nicholas";  
alert(instance.age); // 29
```

# Constructor Stealing

- Methods must be defined inside the constructor, so there's no function reuse.
- Methods defined on the supertype's prototype are not accessible on the subtype, so all types can use only the constructor pattern

# Pseudo-Classical

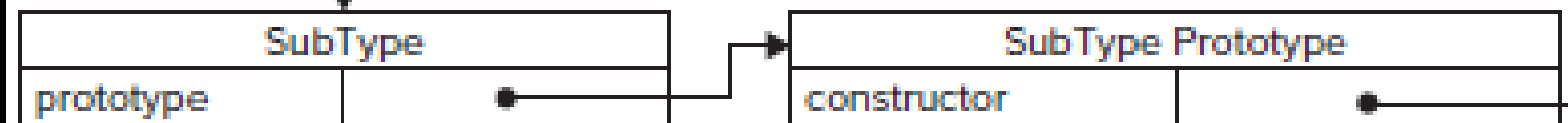
```
function SuperType(name) {
  this.name = name;
  this.colors = ["red", "blue", "green"];
}

SuperType.prototype.sayName = function() {
  alert(this.name);
};

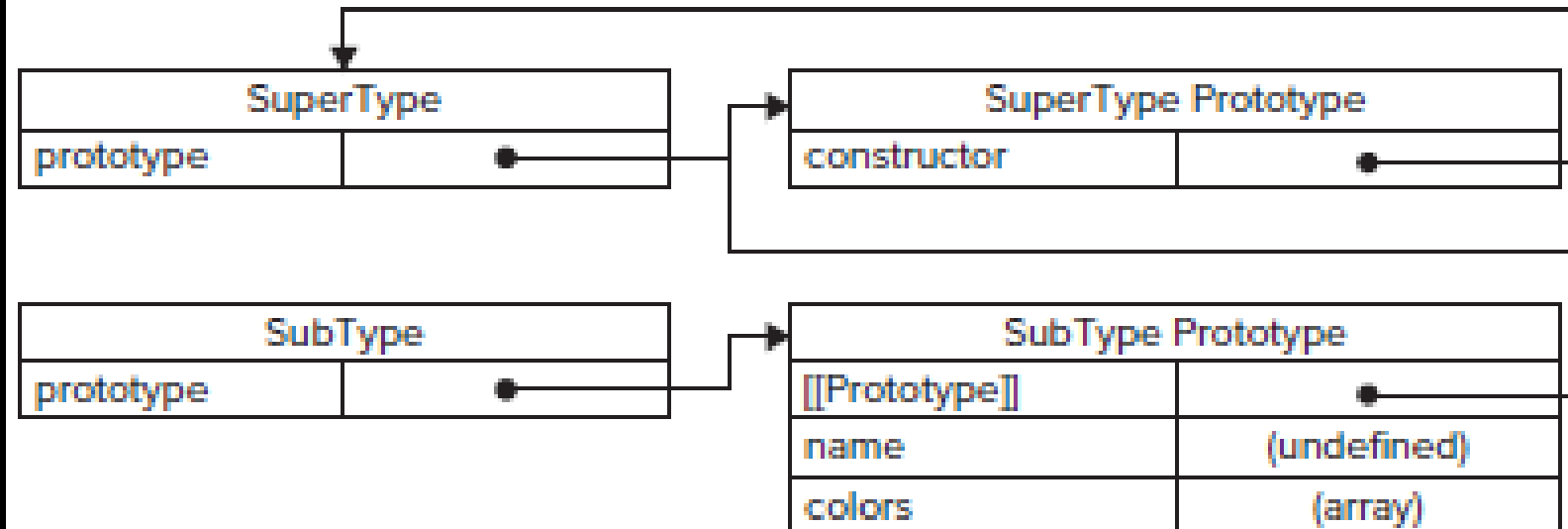
function SubType(name, age) {
  //inherit properties
  SuperType.call(this, name);
  this.age = age;
}

//inherit methods
SubType.prototype = new SuperType();
SubType.prototype.constructor = SubType; // erased by previous line
SubType.prototype.sayAge = function() {
  alert(this.age);
};
```

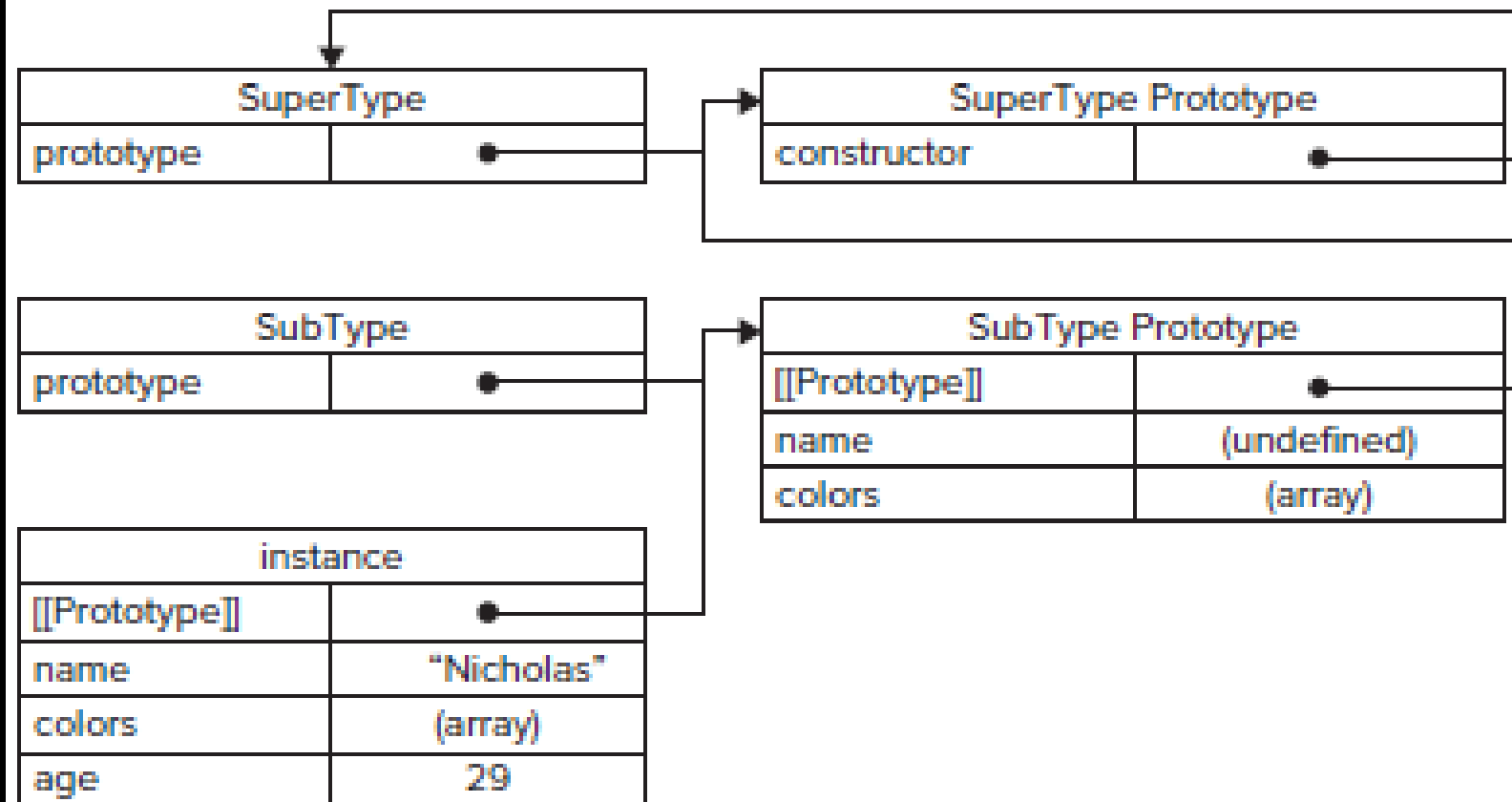
Initially



`SubType.prototype = new SuperType()`



`var instance = new SubType("Nicholas", 29)`



# Pseudo-Classical (other)

```
// Parent
var Mammal = function(name) {
    this.name = name;
}

Mammal.prototype.getName = function() {
    return this.name;
};

Mammal.prototype.says = function() {
    return this.saying || '';
};

// Child
var Cat = function(name) {
    this.name = name;
    this.saying = 'meow';
};
```



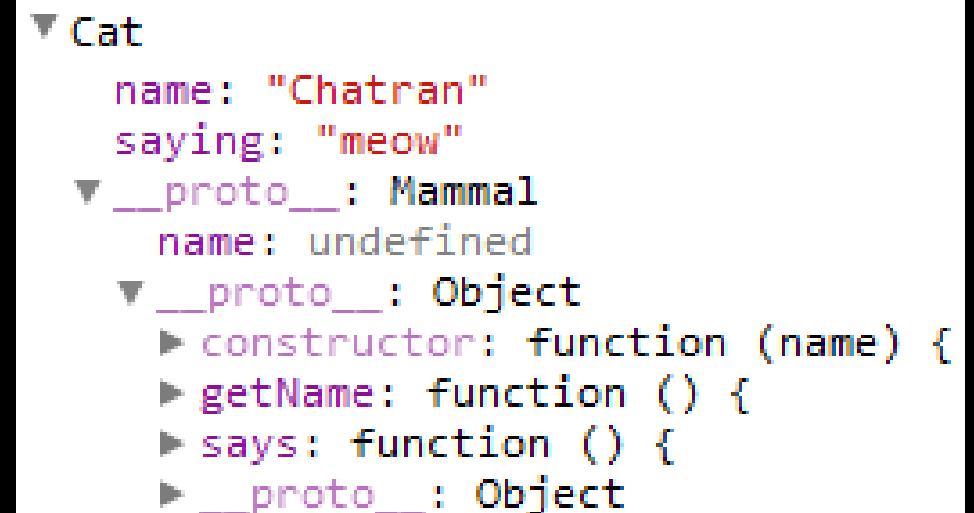
# Pseudo-Classical (other)

```
var inherits = function(child, parent) {  
  child.prototype = new parent();  
  // Why not child.prototype = parent.prototype;?  
};
```

```
inherits(Cat, Mammal);
```

```
var cat = new Cat('Chatran');
```

```
console.log(cat.getName());
```



```
▼ Cat  
  name: "Chatran"  
  saying: "meow"  
  ▼ __proto__: Mammal  
    name: undefined  
    ▼ __proto__: Object  
      ► constructor: function (name) {  
      ► getName: function () {  
      ► says: function () {  
      ► __proto__: Object
```

# Object specifiers

```
// Many arguments
```

```
var myObject = maker(f, l, m, c, s);
```

```
// One argument
```

```
var myObject = maker({  
    first: f,  
    last: l,  
    state: s,  
    city: c  
});
```

# Prototypal

```
var object = function(parent) {  
  var F = function() {};  
  F.prototype = parent;  
  return new F();  
};  
// Or use Object.create(parent);
```

```
var myMammal = {  
  name : 'Herb the Mammal',  
  getName : function() {  
    return this.name;  
  },  
  says : function() {  
    return this.saying || '';  
  }  
};
```

# Prototypal

```
var myCat = object(myMammal);  
// var myCat = Object.create(myMammal)  
  
myCat.name = 'Chatran';  
  
myCat.saying = 'meow';  
  
myCat.getName = function() {  
    return this.says() + ' ' + this.name + ' ' +  
    this.says();  
};  
  
console.log(myCat.getName());
```

```
> console.dir(myCat)  
▼ F  
  ▶ getName: function () {  
    name: "Chatran"  
    saying: "meow"  
  }  
  ▼ __proto__: Object  
    ▶ getName: function () {  
      name: "Herb the Mammal"  
    }  
    ▶ says: function () {  
    }  
    ▶ __proto__: Object
```

# Functional

- AKA Parasitic Inheritance
- Create a function that does the inheritance
- Augments the object in some way
- Return the object as if it did all the work

# Functional

```
var mammal = function (spec) {  
  var that = {};  
  that.get_name = function ( ) {  
    return spec.name;  
  };  
  
  that.says = function ( ) {  
    return spec.saying || '';  
  };  
  
  return that;  
  
};  
  
var myMammal = mammal({name: 'Herb'});
```

# Functional

```
var cat = function (spec) {  
  spec.saying = spec.saying || 'meow';  
  
  var that = mammal(spec);  
  
  that.get_name = function ( ) {  
    return that.says( ) + ' ' + spec.name +  
      ' ' + that.says( );  
  
    return that;  
  
  };  
  
  var myCat = cat({name: 'Henrietta'});
```

# Exercise

Using the functional inheritance, How would you call a method in the parent object?



# Mixin (or Parts)

```
// Our Mixin

var movable = {
  moveUp: function(){ console.log( "move up" ); },
  moveDown: function(){ console.log( "move down" ); },
  stop: function(){ console.log( "stop! in the name of love!" ); }
};

function CarAnimator(){
  this.moveLeft = function(){
    console.log( "move left" );
  };
}

extend( CarAnimator.prototype, movable ); // Extend with our movable Mixin

// Create a new instance of carAnimator

var myAnimator = new CarAnimator();
myAnimator.moveLeft(); // move left
myAnimator.moveDown(); // move down
myAnimator.stop(); // stop! in the name of love!
```

# Mixin

```
function extend(destination, source) {  
  for (var k in source) {  
    if (source.hasOwnProperty(k)) {  
      destination[k] = source[k];  
    }  
  }  
  
  return destination;  
}
```

# Exercise

- Copy all of the properties in the source objects over to the destination object, and return the destination object. It's in-order, so the last source will override properties of the same name in previous arguments

```
utils.extend(destination, *sources)
```

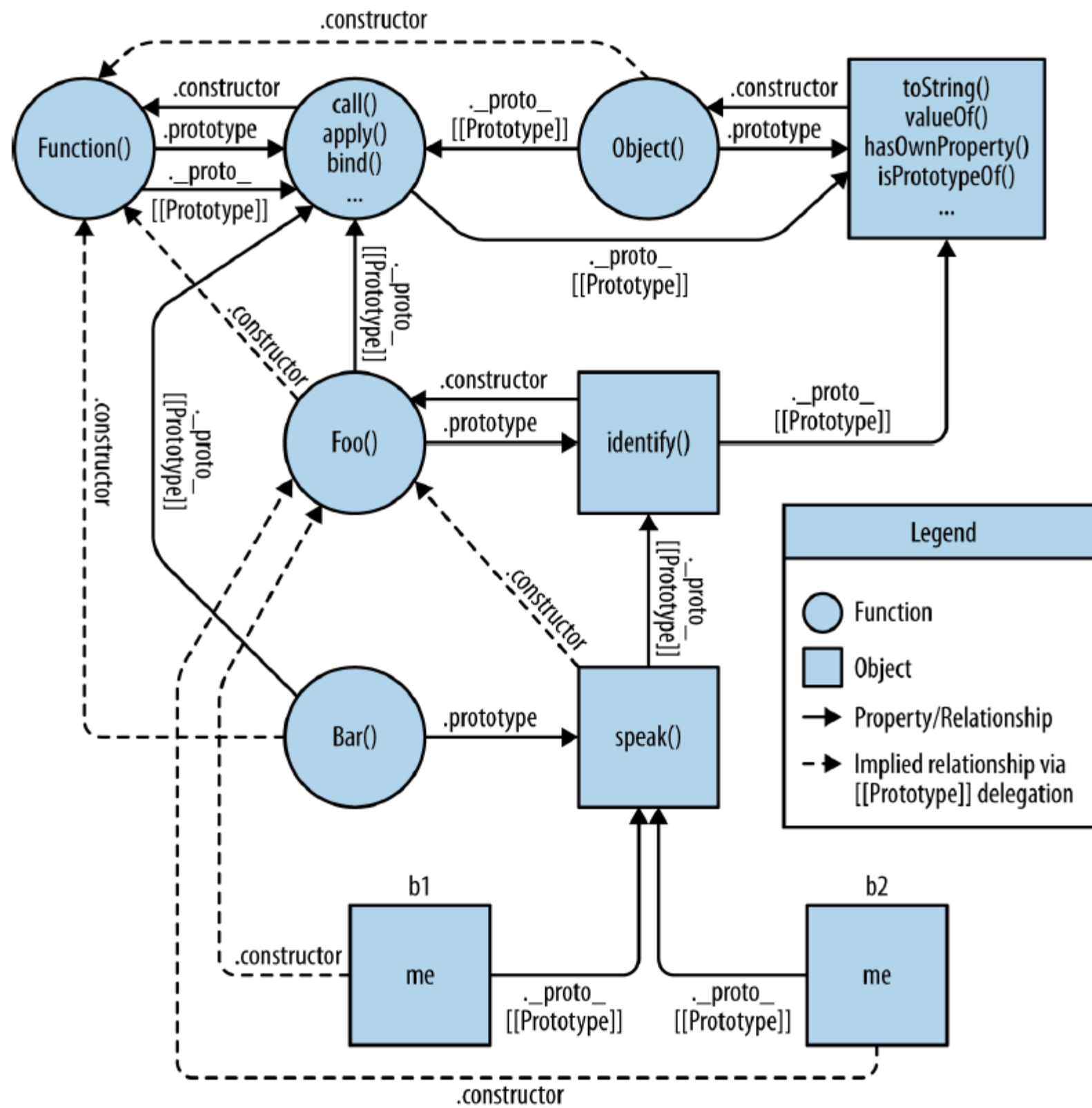
```
utils.extend(destination, mixin1,  
mixin2, ..., mixinN);
```

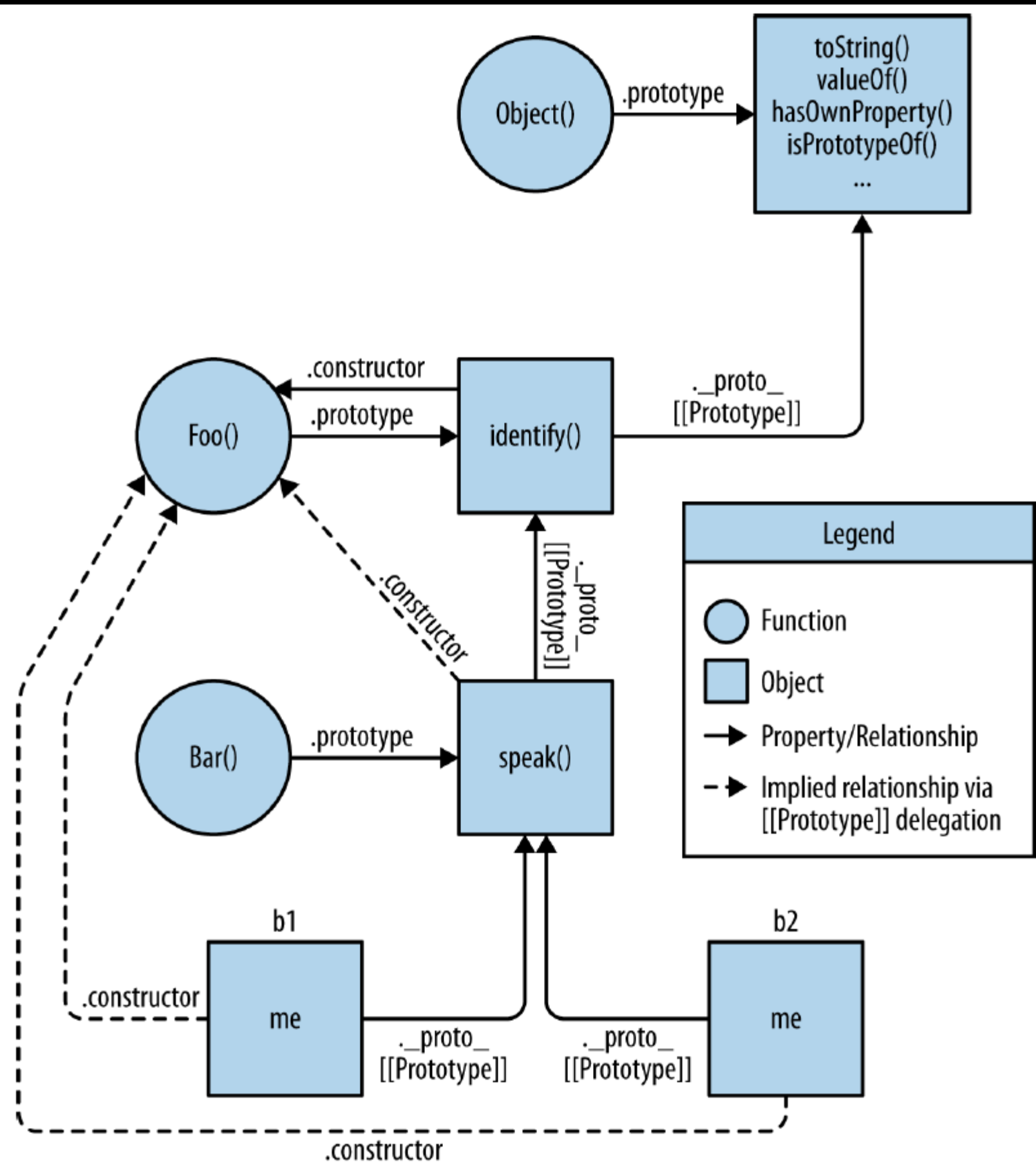
# OLOO

Objects Linked to Other Objects  
Toward Delegation-Oriented Design



```
function Foo(who) {  
    this.me = who;  
}  
Foo.prototype.identify = function() {  
    return "I am " + this.me;  
};  
  
function Bar(who) {  
    Foo.call( this, who );  
}  
Bar.prototype = Object.create( Foo.prototype );  
  
Bar.prototype.speak = function() {  
    alert( "Hello, " + this.identify() + "." );  
};  
  
var b1 = new Bar( "b1" );  
var b2 = new Bar( "b2" );  
  
b1.speak();  
b2.speak();
```

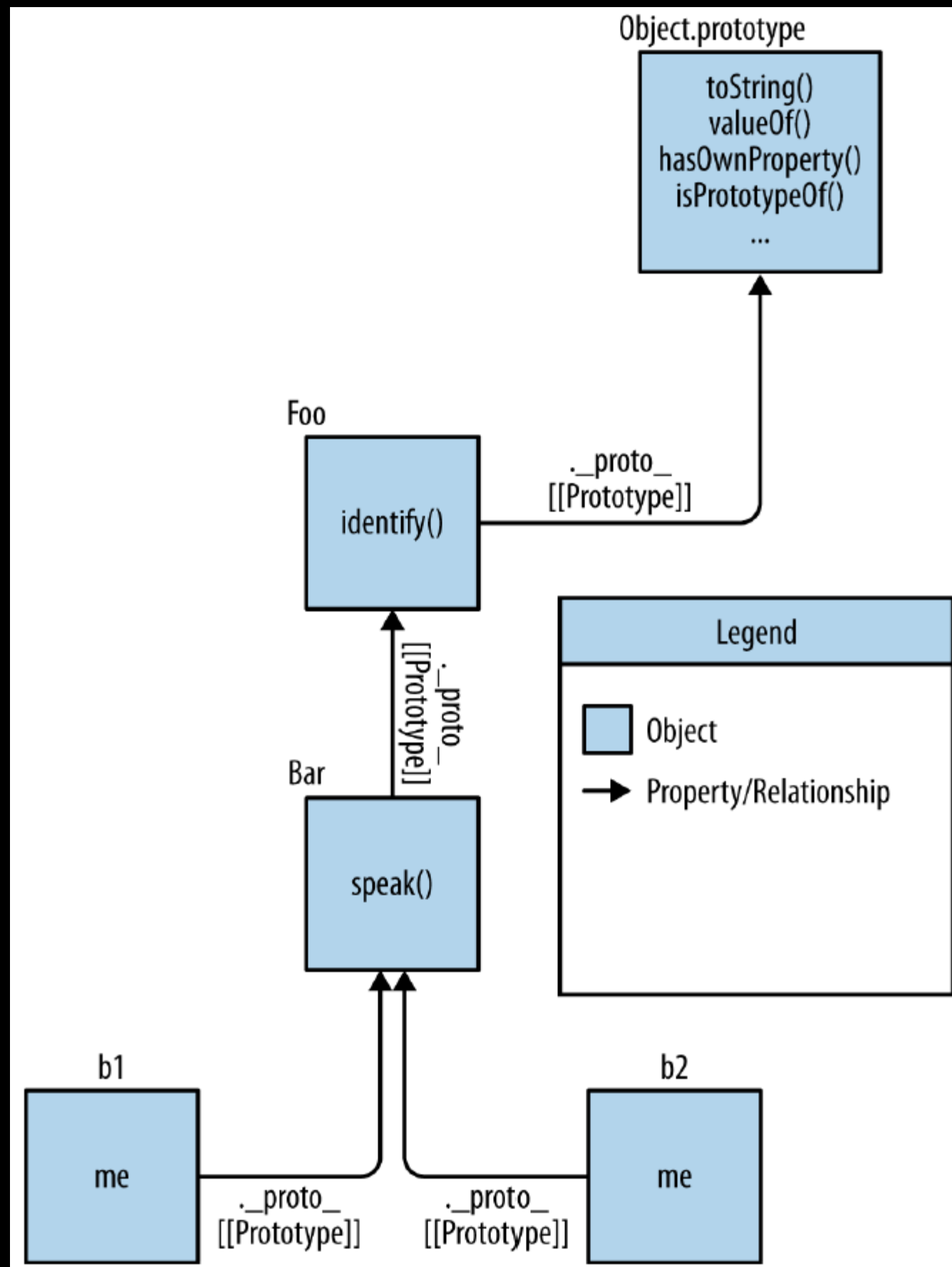




# OLOO

```
Foo = {  
  init: function(who) {  
    this.me = who;  
  },  
  identify: function() {  
    return "I am " + this.me;  
  }  
};  
  
Bar = Object.create( Foo );  
  
Bar.speak = function() {  
  alert( "Hello, " + this.identify() + "." );  
};  
  
var b1 = Object.create( Bar );  
b1.init( "b1" );  
var b2 = Object.create( Bar );  
b2.init( "b2" );  
  
b1.speak();  
b2.speak();
```





# Mixins, Forwarding and Delegation

# Mixins, Forwarding and Delegation

```
var sam = {  
  firstName: 'Sam',  
  lastName: 'Lowry',  
  fullName: function () {  
    return this.firstName + " " + this.lastName;  
  },  
  rename: function (first, last) {  
    this.firstName = first;  
    this.lastName = last;  
    return this;  
  }  
}
```

We can separate its domain properties from its behaviour:

```
var sam = {  
  firstName: 'Sam',  
  lastName: 'Lowry'  
};  
  
var Person = {  
  fullName: function () {  
    return this.firstName + " " + this.lastName;  
  },  
  rename: function (first, last) {  
    this.firstName = first;  
    this.lastName = last;  
    return this;  
  }  
};
```

```
var __slice = [].slice;

function extend () {
    var consumer = arguments[0],
        providers = __slice.call(arguments, 1),
        key,
        i,
        provider;

    for (i = 0; i < providers.length; ++i) {
        provider = providers[i];
        for (key in provider) {
            if (provider.hasOwnProperty(key)) {
                consumer[key] = provider[key];
            }
        };
    };
    return consumer;
};

extend(sam, Person);

sam.rename
    //=> [Function]
```

```
extend(sam, Person);

var peck = {
  firstName: 'Sam',
  lastName: 'Peckinpah'
};

extend(peck, Person);
```

```
var HasCareer = {
  career: function () {
    return this.chosenCareer;
  },
  setCareer: function (career) {
    this.chosenCareer = career;
    return this;
  }
};
```

```
extend(peck, Person);
extend(peck, HasCareer);
```

```
peck.setCareer('Director');
```

# Privacy through closures

```
function HasPrivateCareer (obj) {  
  var chosenCareer;  
  
  obj.career = function () {  
    return chosenCareer;  
  };  
  obj.setCareer = function (career) {  
    chosenCareer = career;  
    return this;  
  };  
  return obj;  
}  
  
HasPrivateCareer(peck);
```

# Privacy through objects

```
function extendPrivately (receiver, mixin) {  
  var methodName,  
      privateProperty = Object.create(null);  
  
  for (methodName in mixin) {  
    if (mixin.hasOwnProperty(methodName)) {  
      receiver[methodName] = mixin[methodName].bind(privateProperty);  
    };  
  };  
  return receiver;  
};
```



```
extendPrivately(twain, HasCareer);  
twain.setCareer( 'Author' );  
twain.career()  
    //=> 'Author'
```

Has it modified twain's properties?

```
twain.chosenCareer  
    //=> undefined
```

## another way to achieve privacy through objects

In our scheme above, we used `.bind` to create methods bound to a private object before mixing references to them into our object. There is another way to do it:

```
function forward (receiver, metaobject, methods) {  
  if (methods == null) {  
    methods = Object.keys(metaobject).filter(function (methodName) {  
      return typeof(metaobject[methodName]) == 'function';  
    });  
  }  
  methods.forEach(function (methodName) {  
    receiver[methodName] = function () {  
      var result = metaobject[methodName].apply(metaobject, arguments);  
      return result === metaobject ? this : result;  
    };  
  });  
  
  return receiver;  
};
```

```
var portfolio = {  
  _investments: [],  
  addInvestment: function (investment) {  
    this._investments.push(investment);  
    return this;  
  },  
  netWorth: function () {  
    return this._investments.reduce(  
      function (acc, investment) {  
        return acc + investment.value;  
      },  
      0  
    );  
  }  
};
```

And next we create an investor who has this portfolio of investments:

```
var investor = {  
  //...  
}
```

What if we want to make investments and to know an investor's net worth?

```
forward(investor, portfolio);
```

# Delegation

|                      | <i>Early-bound</i> | <i>Late-bound</i> |
|----------------------|--------------------|-------------------|
| Receiver's context   | Mixin              |                   |
| Metaobject's context | Private Mixin      | Forwarding        |

## delegation

Let's build it. Here's our `forward` function, modified to evaluate method invocation in the receiver's context:

```
function delegate (receiver, metaobject, methods) {  
  if (methods == null) {  
    methods = Object.keys(metaobject).filter(function (methodName) {  
      return typeof(metaobject[methodName]) == 'function';  
    });  
  }  
  methods.forEach(function (methodName) {  
    receiver[methodName] = function () {  
      return metaobject[methodName].apply(receiver, arguments);  
    };  
  });  
  
  return receiver;  
};
```

# Delegation vs Forwarding

- Delegation and forwarding are both very similar. One metaphor that might help distinguish them is to think of receiving an email asking you to donate some money to a worthy charity.
- If you forward the email to a friend, and the friend donates money, the friend is donating their own money and getting their own tax receipt.

```
consumer[methodName] = function () {  
    return metaobject[methodName].apply(metaobject, arguments);  
}
```

- If you delegate responding to your accountant, the accountant donates your money to the charity and you receive the tax receipt.

```
function () {  
    return metaobject[methodName].apply(receiver, arguments);  
}
```

# Async JavaScript

From Callbacks to Promises

# Async sources

- UI events (click, scroll, mousemove, etc)
- HTTP requests
- setTimeout/setInterval



# Pyramid of Doom

```
step1(function (value1) {  
    step2(value1, function(value2) {  
        step3(value2, function(value3) {  
            step4(value3, function(value4) {  
                // Do something with value4  
            });  
        });  
    });  
});
```

# Handling Async Errors

```
function JSOToOobject(jsonStr) {  
    return JSON.parse(jsonStr);  
}  
var obj = JSOToOobject('{');
```

```
SyntaxError: Unexpected end of input  
    at Object.parse (native)  
    at JSOToOobject (/AsyncJS/stackTrace.js:2:15)  
    at Object.<anonymous> (/AsyncJS/stackTrace.js:4:11)
```

# Handling Async Errors

```
setTimeout(function A() {  
  setTimeout(function B() {  
    setTimeout(function C() {  
      throw new Error('Something terrible has happened!');  
    }, 0);  
  }, 0);  
}, 0);
```

The result of this application is an extraordinarily short stack trace.

```
Error: Something terrible has happened!  
    at Timer.C (/AsyncJS/nestedErrors.js:4:13)
```

Wait a minute—what happened to A and B? Why aren't they in the stack trace? Well, because they weren't on the stack when C ran. Each of the three functions was run directly from the event queue.

# Handling Async Errors

For the same reason, we can't catch errors thrown from async callbacks with a try/catch block. Here's a demonstration:

EventModel/asyncTry.js

```
try {
  setTimeout(function() {
    throw new Error('Catch me if you can!');
  }, 0);
} catch (e) {
  console.error(e);
}
```

# Node Async Errors

- Callbacks in Node.js almost always take an error as their first argument, allowing the callback to decide how to handle it.

```
var fs = require('fs');
fs.readFile('fhgwgdz.txt', function(err, data) {
  if (err) {
    return console.error(err);
  };
  console.log(data.toString('utf8'));
});
```

# Browser Async Errors

- Client-side JavaScript libraries are less consistent, but the most common pattern is for there to be separate callbacks for success and failure. jQuery's Ajax methods follow this pattern.

```
$.get('/data', {  
    success: successHandler,  
    failure: failureHandler  
});
```

# Nested callbacks

```
step1(function(result1) {  
  step2(function(result2) {  
    step3(function(result3) {  
      // and so on...  
    });  
  });  
});
```

```
function checkPassword(username, passwordGuess, callback) {  
  var queryStr = 'SELECT * FROM user WHERE username = ?';  
  db.query(selectUser, username, function (err, result) {  
    if (err) throw err;  
    hash(passwordGuess, function(passwordGuessHash) {  
      callback(passwordGuessHash === result['password_hash']);  
    });  
  });  
}
```

# Un-nesting Callbacks

```
function checkPassword(username, passwordGuess, callback) {  
  var passwordHash;  
  var queryStr = 'SELECT * FROM user WHERE username = ?';  
  db.query(selectUser, username, queryCallback);  
  
  function queryCallback(err, result) {  
    if (err) throw err;  
    passwordHash = result['password_hash'];  
    hash(passwordGuess, hashCallback);  
  }  
  
  function hashCallback(passwordGuessHash) {  
    callback(passwordHash === passwordGuessHash);  
  }  
}
```



Async.js

# Async.js

- Helpers
  - each, map, filter, reduce
- Control flow
  - parallel, series, waterfall

# Quick examples

```
async.map(['file1','file2','file3'], fs.stat, function(err, results){  
    // results is now an array of stats for each file  
});
```

```
async.filter(['file1','file2','file3'], fs.exists, function(results){  
    // results now equals an array of the existing files  
});
```

```
async.parallel([  
    function(){ ... },  
    function(){ ... }  
], callback);
```

```
async.series([  
    function(){ ... },  
    function(){ ... }  
]);
```

# async#each

## **each(arr, iterator, callback)**

Applies the function `iterator` to each item in `arr`, in parallel. The `iterator` is called with an item from the list, and a callback for when it has finished. If the `iterator` passes an error to its `callback`, the main `callback` (for the `each` function) is immediately called with the error.

### **Arguments**

- `arr` - An array to iterate over.
- `iterator(item, callback)` - A function to apply to each item in `arr`. The iterator is passed a `callback(err)` which must be called once it has completed. If no error has occurred, the `callback` should be run without arguments or with an explicit `null` argument.
- `callback(err)` - A callback which is called when all `iterator` functions have finished, or an error occurs.

```
// assuming openFiles is an array of file names

async.each(openFiles, function( file, callback) {

    // Perform operation on file here.
    console.log('Processing file ' + file);

    if( file.length > 32 ) {
        console.log('This file name is too long');
        callback('File name too long');
    } else {
        // Do work to process file here
        console.log('File processed');
        callback();
    }
}, function(err){
    // if any of the file processing produced an error, err would equal that error
    if( err ) {
        // One of the iterations produced an error.
        // All processing will now stop.
        console.log('A file failed to process');
    } else {
        console.log('All files have been processed successfully');
    }
});
```

# async#series vs async#parallel

## **series(tasks, [callback])**

Run the functions in the `tasks` array in series, each one running once the previous function has completed. If any functions in the series pass an error to its callback, no more functions are run, and `callback` is immediately called with the value of the error. Otherwise, `callback` receives an array of results when `tasks` have completed.

## **parallel(tasks, [callback])**

Run the `tasks` array of functions in parallel, without waiting until the previous function has completed. If any of the functions pass an error to its callback, the main `callback` is immediately called with the value of the error. Once the `tasks` have completed, the results are passed to the final `callback` as an array.

### **Arguments**

- `tasks` - An array or object containing functions to run, each function is passed a `callback(err, result)` it must call on completion with an error `err` (which can be `null`) and an optional `result` value.
- `callback(err, results)` - An optional callback to run once all the functions have completed. This function gets a results array (or object) containing all the result arguments passed to the `task` callbacks.

# async#series

```
async.series([
  function(callback){
    // do some stuff ...
    callback(null, 'one');
  },
  function(callback){
    // do some more stuff ...
    callback(null, 'two');
  }
],
// optional callback
function(err, results){
  // results is now equal to ['one', 'two']
});
```

# async#parallel

```
async.parallel([
  function(callback){
    setTimeout(function(){
      callback(null, 'one');
    }, 200);
  },
  function(callback){
    setTimeout(function(){
      callback(null, 'two');
    }, 100);
  }
],
// optional callback
function(err, results){
  // the results array will equal ['one','two'] even though
  // the second function had a shorter timeout.
});
```



# async#waterfall

## **waterfall(tasks, [callback])**

Runs the `tasks` array of functions in series, each passing their results to the next in the array.

However, if any of the `tasks` pass an error to their own callback, the next function is not executed, and the main `callback` is immediately called with the error.

### **Arguments**

- `tasks` - An array of functions to run, each function is passed a `callback(err, result1, result2, ...)` it must call on completion. The first argument is an error (which can be `null`) and any further arguments will be passed as arguments in order to the next task.
- `callback(err, [results])` - An optional callback to run once all the functions have completed. This will be passed the results of the last task's callback.

# async#waterfall

```
async.waterfall([
  function(callback){
    callback(null, 'one', 'two');
  },
  function(arg1, arg2, callback){
    // arg1 now equals 'one' and arg2 now equals 'two'
    callback(null, 'three');
  },
  function(arg1, callback){
    // arg1 now equals 'three'
    callback(null, 'done');
  }
], function (err, result) {
  // result now equals 'done'
});
```

# Promises

with



# Promises

- Callback pyramid
- Promises/A
- Callbacks to Promises
- Chaining Promises
- Flow Control

# What is a promise?

- If a function cannot return a value or throw an exception without blocking, it can return a promise instead.
- A promise is an object that represents the return value or the thrown exception that the function may eventually provide. A promise can also be used as a proxy for a remote object to overcome latency.

# Pyramid of Doom

```
step1(function (value1) {  
    step2(value1, function(value2) {  
        step3(value2, function(value3) {  
            step4(value3, function(value4) {  
                // Do something with value4  
            });  
        });  
    });  
});
```

```
Q.fcall(promisedStep1)
  .then(promisedStep2)
  .then(promisedStep3)
  .then(promisedStep4)
  .then(function (value4) {
    // Do something with value4
  })
  .catch(function (error) {
    // Handle any error from all above steps
  })
  .done();
```

With this approach, you also get implicit error propagation, just like `try`, `catch`, and `finally`. An error in `promisedStep1` will flow all the way to the `catch` function, where it's caught and handled. (Here `promisedStepN` is a version of `stepN` that returns a promise.)

# Promise States

- A promise must be in one of three states: pending, fulfilled, or rejected.
- When pending, a promise:  
may transition to either the fulfilled or rejected state.
- When fulfilled, a promise:  
must not transition to any other state.  
must have a value, which must not change.



# Promise States

- When rejected, a promise:  
must not transition to any other state.  
must have a reason, which must not change.

# The *then* method

- A promise must provide a then method to access its current or eventual value or reason.
- A promise's then method accepts two arguments:  
promise.then(onFulfilled, onRejected)  
Both onFulfilled and onRejected are optional arguments:

# The *then* method

- If `onFulfilled` is a function:  
it must be called after promise is fulfilled, with  
promise's value as its first argument.  
it must not be called before promise is fulfilled.  
it must not be called more than once.
- If `onRejected` is a function,  
it must be called after promise is rejected, with  
promise's reason as its first argument.  
it must not be called before promise is rejected.  
it must not be called more than once.

# The *then* method

- onFulfilled and onRejected must be called as functions (i.e. with no **this** value). [3.2]
- then may be called multiple times on the same promise.  
If/when promise is fulfilled, all respective onFulfilled callbacks must execute in the order of their originating calls to then.  
If/when promise is rejected, all respective onRejected callbacks must execute in the order of their originating calls to then.
- then must return a promise [3.3].  
promise2 = promise1.then(onFulfilled, onRejected);

# Propagation

```
var outputPromise = getInputPromise()  
.then(function (input) {  
}, function (reason) {  
});
```

The `outputPromise` variable becomes a new promise for the return value of either handler. Since a function can only either return a value or throw an exception, only one handler will ever be called and it will be responsible for resolving `outputPromise`.

- If you return a value in a handler, `outputPromise` will get fulfilled.
- If you throw an exception in a handler, `outputPromise` will get rejected.
- If you return a **promise** in a handler, `outputPromise` will “become” that promise. Being able to become a new promise is useful for managing delays, combining results, or recovering from errors.

# Chaining

```
return getUsername()
.then(function (username) {
  return getUser(username)
  .then(function (user) {
    // if we get here without an error,
    // the value returned here
    // or the exception thrown here
    // resolves the promise returned
    // by the first line
  })
});
```

```
return getUsername()
.then(function (username) {
  return getUser(username);
})
.then(function (user) {
  // if we get here without an error,
  // the value returned here
  // or the exception thrown here
  // resolves the promise returned
  // by the first line
});
```

The only difference is nesting. It's useful to nest handlers if you need to capture multiple input values in your closure.

```
function authenticate() {  
  return getUsername()  
    .then(function (username) {  
      return getUser(username);  
    })  
    // chained because we will not need the user name in the next event  
    .then(function (user) {  
      return getPassword()  
        // nested because we need both user and password next  
        .then(function (password) {  
          if (user.passwordHash !== hash(password)) {  
            throw new Error("Can't authenticate");  
          }  
        });  
      });  
    });  
}
```



# Combination

You can turn an array of promises into a promise for the whole, fulfilled array using `all`.

```
return Q.all([
  eventualAdd(2, 2),
  eventualAdd(10, 20)
]);
```

# Combination

The `all` function returns a promise for an array of values. When this promise is fulfilled, the array contains the fulfillment values of the original promises, in the same order as those promises. If one of the given promises is rejected, the returned promise is immediately rejected, not waiting for the rest of the batch. If you want to wait for all of the promises to either be fulfilled or rejected, you can use `allSettled`.

```
Q.allSettled(promises)
  .then(function (results) {
    results.forEach(function (result) {
      if (result.state === "fulfilled") {
        var value = result.value;
      } else {
        var reason = result.reason;
      }
    });
  });
```

# Handling errors

One sometimes-unintuitive aspect of promises is that if you throw an exception in the fulfillment handler, it will not be caught by the error handler.

```
return foo()  
  .then(function (value) {  
    throw new Error("Can't bar.");  
  }, function (error) {  
    // We only get here if "foo" fails  
  });
```

# Handling errors

To see why this is, consider the parallel between promises and `try/catch`. We are `try`-ing to execute `foo()`; the error handler represents a `catch` for `foo()`, while the fulfillment handler represents code that happens *after* the `try/catch` block. That code then needs its own `try/catch` block.

In terms of promises, this means chaining your rejection handler:

```
return foo()
  .then(function (value) {
    throw new Error("Can't bar.");
  })
  .fail(function (error) {
    // We get here with either foo's error or bar's error
  });
```

# Creating promises

## Using Deferreds

If you have to interface with asynchronous functions that are callback-based instead of promise-based, Q provides a few shortcuts (like `Q.nfcall` and friends). But much of the time, the solution will be to use *deferreds*.

```
var deferred = Q.defer();
FS.readFile("foo.txt", "utf-8", function (error, text) {
  if (error) {
    deferred.reject(new Error(error));
  } else {
    deferred.resolve(text);
  }
});
return deferred.promise;
```

```
function requestOkText(url) {
    var request = new XMLHttpRequest();
    var deferred = Q.defer();

    request.open("GET", url, true);
    request.onload = onload;
    request.onerror = onerror;
    request.onprogress = onprogress;
    request.send();

    function onload() {
        if (request.status === 200) {
            deferred.resolve(request.responseText);
        } else {
            deferred.reject(new Error("Status code was " + request.status));
        }
    }

    function onerror() {
        deferred.reject(new Error("Can't XHR " + JSON.stringify(url)));
    }

    function onprogress(event) {
        deferred.notify(event.loaded / event.total);
    }

    return deferred.promise;
}
```

Below is an example of how to use this `requestOkText` function:

```
requestOkText("http://localhost:3000")
.then(function (responseText) {
    // If the HTTP response returns 200 OK, log the response text.
    console.log(responseText);
}, function (error) {
    // If there's an error or a non-200 status code, log the error.
    console.error(error);
}, function (progress) {
    // Log the progress as it comes in.
    console.log("Request progress: " + Math.round(progress * 100) + "%");
});
```

# Q.defer: Interject Promise Support

- Use when you have to intermingle other logic inside callback work

```
function getLocation() {
  var deferred = Q.defer();

  console.log("Calling getCurrentPosition");
  navigator.geolocation.getCurrentPosition(function(position) {
    deferred.resolve(position);
    console.log("getCurrentPosition resolved");
  }, function(error){
    deferred.reject(error);
    console.log("getCurrentPosition errored");
  });

  return deferred.promise;
};
```



# Exercise

- Create a promise using Q, that will resolve fn after x milliseconds

```
utils.delay(fn, ms).then(function(fnResult) {  
    console.log('called after', ms);  
    console.log('fn result is', fnResult);  
})
```

The End

# Bibliography

- 2008 O'Reilly - JavaScript: The Good Parts by Douglas Crockford
- 2012 Wrox - Professional JavaScript for Web Developers 3rd Edition - Nicholas C. Zakas
- 2012 O'Reilly - Learning JavaScripts Design Patterns - Addy Osmani  
<http://addyosmani.com/resources/essentialjsdesignpatterns/book/https://github.com/addyosmani/essential-js-design-patterns>

# Links

- <http://raganwald.com/2014/04/10/mixins-forwarding-delegation.html>
- <http://davidwalsh.name/javascript-objects-deconstruction>
- <https://github.com/caolan/async>
- <http://promisesaplus.com/>
- <http://documentup.com/kriskowal/q/>

# Presenter Contact Info

- Matias Volpe
- [matias.volpe@arrisi.com](mailto:matias.volpe@arrisi.com)