JavaScript - Objects and Prototypes

CPEN322

The University of British Columbia
Department of Electrical and Computer Engineering
Vancouver, Canada

ece Electrical and Computer Engineering

UBC

February 7, 2024

Javascript: History and Philosophy

1. Javascript: History and Philosophy
2. Object Creation in Javascript
3. Object Constructor and Methods
4. Prototypes and Inheritance
5. Type-Checking and Reflection

Javascript: History

- Invented in 10 days by Brendan Eich at Nescape in May 1995 as part of the Navigator 2.0 browser
  - Based on Self, but dressed up to look like Java
  - Standardized by committee in 2000 as ECMAScript

> **Brendan Eich (Inventor of JavaScript):**
>
> *JavaScript (JS) had to "look like Java" only less so, be Java's dumb kid brother or boy-hostage sidekick. Plus, I had to be done* ***in ten days*** *or something worse than JS would have happened*

Exercise

```
var j=6;
function foo() {
  var j;
  j=7;
}

foo();
window.alert(j); // what is the value of j?
```

Numbers

- a single **Number** type, represented internally as a 64-bit floating point (similar to double in Java)
- $1 + 2 = 3$
- 0.1 + 0.2 !== 03 (it's 0.30000000000000004 so be careful when using floats)

```
var result = (0.1 + 0.2).toFixed(1); // "0.3" as string
result = Number(result); // Convert to a number
```

- **NaN** (Not-a-Number): A special value that indicates an unrepresentable or undefined result, such as the result of dividing 0 by 0

https://javascript.info/number

Javascript: Philosophy

- Everything is an object
  - Includes functions, non-primitive types etc.
  - Even the class of an object is an object !
- Nothing has a type
  - Or its type is what you want it to be (duck typing)
  - No compile-time checking (unless in strict mode)
  - Runtime type errors can occur
- Programmers make mistakes anyways
  - If an exception is thrown, do not terminate program (artifact of browsers, rather than JS)
- Code is no different from data
  - So we can use ‘eval’ to convert data to code
- Function’s can be called with fewer or more arguments than needed (variadic functions)

Duck Typing (dynamic typing)

The term comes from the phrase "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."

```
function makeItQuack(duck) {
  if (duck.quack) {
    duck.quack();
  }
  else {
    console.log("This is not a duck!");
  }
}
const duck = {
  quack: function() {
    console.log("Quack!");
  } };

const dog = {
  bark: function() {
    console.log("Woof!");
} };

makeItQuack(duck); // Outputs: Quack!
makeItQuack(dog);   // Outputs: This is not a duck!
```

This lecture

- We'll learn about Objects and Classes the "old way" (without ES6)
- ES6 makes it much simpler to declare and use objects, but ....
- it's just syntactic sugar around the old way of doing things
- Not understanding objects from the ground up can result in nasty surprises
- Things will make a lot more sense if we go from the old way

Object Creation in Javascript

1. Javascript: History and Philosophy
2. Object Creation in Javascript
3. Object Constructor and Methods
4. Prototypes and Inheritance
5. Type-Checking and Reflection

What is an Object in JS ?

- Container of properties, where each property has a name and a value, and is mutable
  - Property names can be any string, including the empty string
  - Property values can be anything except undefined
- What are not objects ?
  - Primitive types such as numbers, booleans, strings
  - null and undefined – these are special types

What about classes ?

- There are no classes in JavaScript, as we understand them in languages such as Java
- "What ? How can we have objects without classes ?"
  - Objects use what are known as prototypes
  - An object can inherit the properties of another object using prototype linkage (more later)

Object Creation via Object Literals

```
// Initializing an empty object
var empty_object = {};

// Object with two attributes
var name = {
    firstName: "John",
    lastName: "Doe"
};
```

NOTE

You don't need a quote around firstName and lastName as they're valid JavaScript identifiers

Retrieving an Object's Property

```
name["firstName"]
// Equivalent to:
name.firstName

name["lastName"]
// Equivalent to:
name.lastName
```

- What if you write `name["middleName"]`?
  - Returns undefined. Later use of this value will result in an "TypeError" exception being thrown

Update of an Object's Property

```
name["firstName"] = "Different firstName";
name.lastName = "Different lastName";
```

- What happens if the property is not present ?
  - It'll get **added** to the object with the value!
- In short, objects behave like hash tables in JS

Objects are passed by REFERENCE !

- In JavaScript, objects are passed by REFRENCE
  - No copies are ever made unless explicitly done/asked
    - i.e., JSON.parse(JSON.stringify(obj))
  - Changes made in one instance are instantly visible in all instances as it is by reference

JSON.parse creates a copy of obj

```
let obj = {   name: "John",   age: 30 };

let jsonString = JSON.stringify(obj);

console.log(jsonString); //Outputs: '{"name":"John","age":30}'

console.log(obj);        //Outputs: { name: "John", age: 30 }
```

Object Constructor and Methods

1 Javascript: History and Philosophy

2 Object Creation in Javascript

3 Object Constructor and Methods

4 Prototypes and Inheritance

5 Type-Checking and Reflection

Object Creation via Constructor Functions

- Define the object type by writing a Constructor Function
  - By convention, use a capital letter as first letter for the object name
  - Use “**this**” within function to initialize properties
- Call constructor function with the **new** operator and pass it the values to initialize
  - Forgetting the ‘new’ can have unexpected effects
- ‘new’ operator to create an object of instance ‘Object’, which is a global, unique JavaScript object

Object Creation using New

```
var Person = function(firstName, lastName, gender) {
    this.firstName= firstName;
    this.lastName = lastName;
    this.gender = gender;
}
var p = new Person("John", "Smith", "Male");
```

Good practice to avoid forgetting “new”

‘new’ operator to create

Object Creation using New

```
var Person = function(firstName, lastName, gender) {
  if (!(this instanceof Person)) {
    throw new Error("Person must be called with the new keyword");
  }
  this.firstName = firstName;
  this.lastName = lastName;
  this.gender = gender;
}
```

this keyword

- It's a reference to the current object, and is valid only inside the object
- Need to explicitly use **this** to reference the object's fields and methods
    - Forgetting this means you'll create new local vars
    - Can be stored in ordinary local variables

Constructors

- Using the new operator as we've seen
- this is set to the new object that was created
  - Automatically returned unless the constructor chooses to return another object (non-primitive)
- Bad things can happen if you forget the 'new' before the call to the constructor

What is the value of p.name?

missing New

```
function Person(name) {
  this.name = name;
}
var p = Person("John");
console.log(p.name);
```

Object Methods

- Functions that are associated with an object
- Like any other field of the object and invoked as object.methodName()
    - Example: person.fullName();
    - this is automatically defined inside the method

```
var Person = function(firstName, lastName) {
        this.firstName= firstName;
        this.lastName = lastName;
        fullName: function() {
        return this.firstName + " " + this.lastName;
    }
}
var person = new Person("John", "Doe");
console.log(person.fullName()); // Output: "John Doe"
```

NOTE

this is bound to the object on which it is invoked

Calling a Method

- Simply say object.methodName(parameters)
- Example: person.fullName();
- this is bound to the object on which it is called. In the example, this = person. This binding occurs at invocation time (late binding).

Object creation via Object.create()

- **Object.create** creates an object from another existing object
- Example: jane = Object.create(person);
- The Object.create() method creates a new object, using an existing object as the **prototype** of the newly created object.

Prototypes and Inheritance

1. Javascript: History and Philosophy
2. Object Creation in Javascript
3. Object Constructor and Methods
4. Prototypes and Inheritance
5. Type-Checking and Reflection

Object Prototype

- Every object has a field called Prototype
  - Prototype is a pointer to the object the object is created from
  - Changing the **prototype object** instantly changes all instances of the object
- The default prototype value for a given object is Object
  - Can be changed when using new or Object.create to construct the object
- Object has null as its prototype. null is the end of the prototype chain.

Object Prototype: Example

- what is the prototype value of a “Person” object ?

```
var p = new Person("John", "Smith", "Male");
console.log( Object.getPrototypeOf(p) );
```

- What will happen if we do the following instead

```
console.log( Object.getPrototypeOf(Person) );
```

Prototype

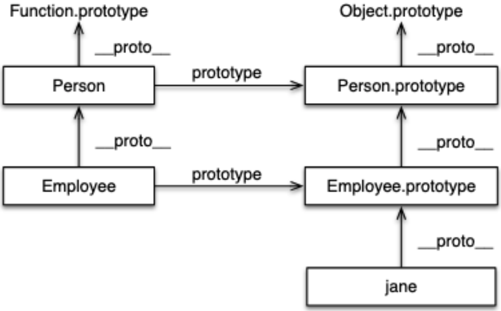
- what is the prototype value of a “Person” object ?

```
var p = new Person("John", "Smith", "Male");
console.log( Object.getPrototypeOf(p) ); // Person.prototype
```

- What will happen if we do the following instead

```
console.log( Object.getPrototypeOf(Person) ); // Function.
    prototpe
```

Prototype Inheritance



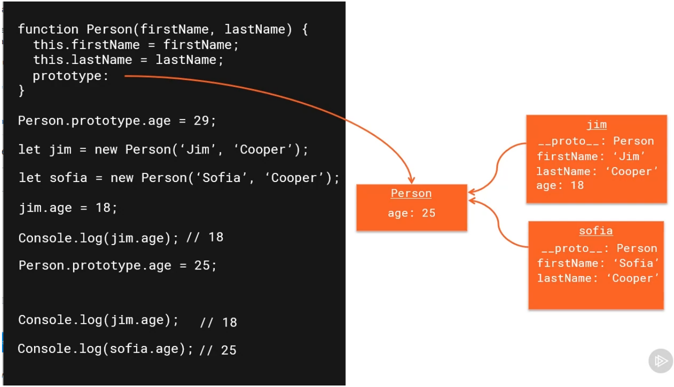
Prototype Example

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

Person.prototype.age = 29;
let jim = new Person('Jim', 'Cooper');
let sofia = new Person('Sofia', 'Cooper');

jim.age = 18;
console.log(jim.age); // ?

Person.prototype.age = 25;

console.log(jim.age);  // ?
console.log(sofia.age); // ?
```

Prototype Example

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

Person.prototype.age = 29;
let jim = new Person('Jim', 'Cooper');
let sofia = new Person('Sofia', 'Cooper');

jim.age = 18;
console.log(jim.age); // 18

Person.prototype.age = 25;

console.log(jim.age);  // 18
console.log(sofia.age); // 25
```

Prototype Example



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

Person.prototype.age = 29;

let jim = new Person(‘Jim’, ‘Cooper’);

let sofia = new Person(‘Sofia’, ‘Cooper’);

jim.age = 18;

Console.log(jim.age); // 18

Person.prototype.age = 25;


Console.log(jim.age);   // 18

Console.log(sofia.age); // 25
```

What 'new' really does?

- Initializes a new native object
- Sets the object's "prototype" field to the constructor function's prototype field
  - In Chrome (V8 engine), the prototype of an object instance o is accessible through the hidden property o.__proto__.
    - Direct usage should be avoided! Use instead Object.getPrototypeOf(o)
  - If it's not an Object, sets it to Object.prototype
    - i.e., Object.create(null)
- Calls the constructor function, with the object as this
  - Any fields initialized by the function are added to this

Prototype Modification

- An object's prototype object is just another object (typically). So it can be modified too.
- We can add properties to prototype objects – the property becomes instantly visible in all instances of that prototype (even if they were created before the property was added)
  - Reflects in all descendant objects as well (later)

Prototype Modification: Example

```
var p1 = new Person("John", "Smith", "Male");

Person.prototype.print = function() {
    console.log( "Person: " + this.firstName
            + this.lastName + this.gender + "\n");
}

var p2 = new Person("Linda", "James", "Female");
p1.print();
p2.print();
```

Delegation with Prototypes

- When you lookup an Object's property, and the property is not defined in the Object,
  - It checks if the Object's prototype is a valid object
  - If so, it does the lookup on the prototype object
  - If it finds the property, it returns it
  - Otherwise, it recursively repeats the above process till it encounters Object.prototype
  - If it doesn't find the property even after all this, it returns undefined

Prototype Inheritance

- Due to Delegation, Prototypes can be used for (simulating) inheritance in JavaScript
  - Set the prototype field of the child object to that of the parent object
  - Any access to child object's properties will first check the child object (so it can over-ride them)
  - If it can't find the property in the child object, it looks up the parent object specified in prototype
  - This process carries on recursively till the top of the prototype chain is reached (Object.prototype)

Exercise: Implement Employee

Implement an Employee that **inherits** from Person

Employee has:

- firstName, lastName, gender, and title

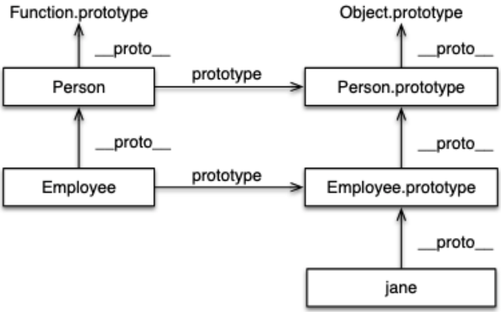
Person has:

- firstname, lastName, gender

Solution: Implement Employee

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

var Employee = function(firstName, lastName, gender, title)
     {
   Person.call( this, firstName, lastName, gender );
   this.title = title;
}

Employee.prototype = new Person();
   /* Why should you create a new person object ? */

Employee.prototype.constructor = Employee;

var jane = new Employee("Jane", "Doe", "Female", "Manager");
```

Visualized


Function.prototype
Object.prototype
__proto__
__proto__
Person
prototype
Person.prototype
__proto__
__proto__
Employee
prototype
Employee.prototype
__proto__
jane

Object.create(proto)

- Creates a new object with the specified prototype object and properties
- proto parameter must be null or an object
  - Throws TypeError otherwise

Object.create Argument

- Can add/specify initialization parameters directly in Object.create as an (optional) 2nd argument

```
var e = Object.create( Person, { title: {value: "Manager" }} )
```

Prototype Inheritance with *Object.create*: Example

```
var Person = {
    firstName:  "John";
    lastName: "Smith";
    gender: "Male";
    print : function() {
        console.log( "Person : " + this.firstName
                     + this.lastName + this.gender;
    }
};
var e = Object.create( Person );
e.title = "Manager";
```

Design Tips

- Object.create might be cleaner in some situations, rather than using new and .prototype (no need for artificial objects)
- With new, you need to remember to use this and also NOT return an object in the constructor
  - Otherwise, bad things can happen
- Object.create allows you to create objects without running their constructor functions
  - Need to run your constructor manually if you want
  - i.e., Person.call(p2, “Bob”)

Class Activity

- Construct a class hierarchy with the following properties:
- Add an area method and a toString prototype function to all the objects.

Point { x, y } ⇒ Circle { x, y ,r } ⇒ Ellipse { x, y, r, r2 }

Class Activity

Start with:

```
// Base Point constructor function
function Point(x, y) {
  this.x = x;
  this.y = y;
}

// Adding toString method to Point prototype
Point.prototype.toString = function() {
  return ‘Point at (${this.x}, ${this.y})‘;
};
```

Solution: Circle

```
// Circle constructor inheriting from Point
function Circle(x, y, r) {
  Point.call(this, x, y); // Call the parent constructor
  this.r = r;
}

// Inheriting from Point prototype
Circle.prototype = Object.create(Point.prototype);
Circle.prototype.constructor = Circle;

// Adding area method to Circle prototype
Circle.prototype.area = function() {
  return Math.PI * this.r * this.r;
};

// Adding toString method to Circle prototype
Circle.prototype.toString = function() {
  return ‘Circle at (${this.x}, ${this.y}) with radius ${
      this.r}‘;
};
```

Solution: Eclipse

```
// Ellipse constructor inheriting from Circle
function Ellipse(x, y, r, r2) {
  Circle.call(this, x, y, r); // Call the parent constructor
  this.r2 = r2;
}

// Inheriting from Circle prototype
Ellipse.prototype = Object.create(Circle.prototype);
Ellipse.prototype.constructor = Ellipse;

// Adding area method to Ellipse prototype
Ellipse.prototype.area = function() {
  return Math.PI * this.r * this.r2;
};

// Adding toString method to Ellipse prototype
Ellipse.prototype.toString = function() {
  return ‘Ellipse at (${this.x}, ${this.y}) with radii ${this.r} and ${this.r2}‘;
};
```

Solution: Usage

```
// Usage
var p = new Point(1, 2);
var c = new Circle(1, 2, 3);
var e = new Ellipse(1, 2, 3, 4);

console.log(p.toString()); // Point at (1, 2)
console.log(c.toString()); // Circle at (1, 2) with radius 3
console.log(e.toString()); // Ellipse at (1, 2) with radii 3 and 4

console.log(c.area()); // 28.274333882308138
console.log(e.area()); // 37.69911184307752
```

Type-Checking and Reflection

1. Javascript: History and Philosophy
2. Object Creation in Javascript
3. Object Constructor and Methods
4. Prototypes and Inheritance
5. Type-Checking and Reflection

Reflection and Type-Checking

- In JS, you can query an object for its type, prototype, and properties at runtime
  - To get the Prototype: getPrototypeOf()
  - To get the type of: typeof
    - "undefined", "boolean", "number", "string", "symbol", "object", "function"
  - To check if it's of certain instance: instanceof
  - To check if it has a certain property: in
  - To check if it has a property, and the property was not inherited through the prototype chain: hasOwnProperty()

typeof

- Can be used for both primitive types and objects

```
typeof( Person.firstName ) => String
typeof( Person.lastName ) => String
typeof( Person.age ) => Number
typeof(Person.constructor) => function (prototype)
typeof(Person.toString) => function (from Object)
typeof(Person.middleName) => undefined
typeof(Person) => object
typeof(null) => object (bug in js!!!)
```

instanceof

- Checks if an object has in its prototype chain the prototype property of the constructor

```
object instanceof constructor => Boolean

// Example:
var p = new Person( /* ... */ );
var e = new Employee( /* ... */ );

p instanceof Person;     // True
p instanceof Employee;    // False
e instanceof Person;   // True
e instanceof Employee;    // True
p instanceof Object;     // True
e instanceof Object;     // True
```

When to use which?

- Use **typeof** when you need to know the **type of a primitive**.
- Use **instanceof** when you need to confirm the **prototype-based inheritance.**

getPrototypeOf

- Gets an object's prototype (From the prototype field) – Object.getPrototypeOf(Obj)
    - Equivalent of 'super' in languages like Java

in operator

- Tests if an object o has property p
    - Checks both object and its prototype chain

```
var p = new Person( /* ... */ );
var e = new Employee( /* ... */ );

"firstName" in p;  // True
"lastName" in e;   // True
"Title" in p;   // False
"Title" in e;   // True
```

hasOwnProperty

- Only checks the object's properties itself
    - Does not follow the prototype chain
    - Useful to know if an object has overridden a property or introduced a new one

```
var p = new Employee( /* ... */ );
p.hasOwnProperty("Title")  // True
p.hasOwnProperty("FirstName")  // False
```

Iterating over an Object's fields

- Go over the fields of an object and perform some action(s) on them (e.g., print them)
  - Can use hasOwnProperty as a filter if needed

```
var name;
for (name in obj) {
    if ( typeof( obj[name] ) != "function") {
        document.writeln(name + " : " + obj[name]);
    }
}
```

Removing an Object's Property

- To remove a property from an object if it has one (not removed from its prototype), use:

```
delete object.property−name
```

- Properties inherited from the prototype cannot be deleted unless the object had overriden them.

```
var e = new Employee( /* ... */ );
delete e.Title;     // Title is removed from e
```

Object Property Types

- Properties of an object can be configured to have the following attributes (or not):
  - Enumerable: Show up during enumeration(for.. in)
  - Configurable: Can be removed using delete, and the attributes can be changed after creation
  - Writeable: Can be modified after creation
- By default, all properties of an object are enumerable, configurable and writeable

Specifying Object Property types

- Can be done during Object creation with Object.create

```
var Person = { ... };

var jane = Object.create(Person, {
  title: {
    value: "Manager",
    enumerable: true,
    configurable: true,
    writable: false
  }
});
```

- Can be done after creation using Object.defineProperty

Design Guidelines

- Use for. . . in loops to iterate over object's properties to make the code extensible
  - Avoid hardcoding property names if possible
  - Use instanceof rather than getPrototypeOf
- Try to fix the attributes of a property at object creation time. With very few exceptions, there is no need to change a property's attribute.

Class Activity

- Write a function to iterate over the properties of a given object, and identify those properties that it **inherited** from its prototype AND **overrode** it with its own values
    - Do not consider functions

Solution

```
function findOverriddenProperties(obj) {
  var overridden = [];
  var currentProto = Object.getPrototypeOf(obj);

  while(currentProto && currentProto !== Object.prototype) {
    for(var prop in currentProto) {
      if (!currentProto.hasOwnProperty(prop)) continue;
      // Check if it's not a function, the property exists
          on obj,
      // and its value is different from that on the
          prototype
      if(typeof currentProto[prop] !== 'function' &&
            obj.hasOwnProperty(prop) && obj[prop] !==
                currentProto[prop]) {
        overridden.push(prop);
      }
    }
    currentProto = Object.getPrototypeOf(currentProto);
  }

  return overridden;
}
```

ES6 Classes

With the introduction of ES6 classes, the syntax for creating prototypes becomes much cleaner, although under the hood, it’s still using the same prototype-based inheritance:

```
class Person {
  constructor(name) {
    this.name = name;
  }
  greet() {
    console.log(’Hello, my name is ’ + this.name);
  }
}
const bob = new Person(’Bob’);
bob.greet(); // Hello, my name is Bob
```

ES6 Extends

```
class Person {
  // Person methods and properties
}
class Employee extends Person {
  // Employee methods and properties
}
// Create an instance of Employee
const jane = new Employee();
```

Table of Contents