Javascript: History and Philosophy

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


```
1  var j=6;
2  function foo() {
3    var j;
4    j=7;
5  }
6
7  foo();
8  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."

```
1  function makeItQuack(duck) {
2    if (duck.quack) {
3      duck.quack();
4    }
5    else {
6      console.log("This is not a duck!");
7    }
8  }
9  const duck = {
10   quack: function() {
11     console.log("Quack!");
12   } };
13
14 const dog = {
15   bark: function() {
16     console.log("Woof!");
17 } };
18
19 makeItQuack(duck); // Outputs: Quack!
20 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

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


```
1  // Initializing an empty object
2  var empty_object = {};
3
4  // Object with two attributes
5  var name = {
6      firstName: "John",
7      lastName: "Doe"
8  };
```

NOTE

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

Retrieving an Object's Property


```
1  name["firstName"]
2  // Equivalent to:
3  name.firstName
4
5  name["lastName"]
6  // Equivalent to:
7  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


```
1  name["firstName"] = "Different firstName";
2  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


```
1  let obj = {   name: "John",   age: 30 };
2
3  let jsonString = JSON.stringify(obj);
4
5  console.log(jsonString); //Outputs: '{"name":"John","age
       ":30}'
6
7  console.log(obj);        //Outputs: { name: "John", age: 30
       }
```

Object Constructor and Methods

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

```
1  var Person = function(firstName, lastName, gender)
       {
2        this.firstName= firstName;
3        this.lastName = lastName;
4        this.gender = gender;
5  }
6  var p = new Person("John", "Smith", "Male");
```

Good practice to avoid forgetting "new"

'new' operator to create

Object Creation using New

```
1  var Person = function(firstName, lastName, gender)
       {
2    if (!(this instanceof Person)) {
3      throw new Error("Person must be called with
           the new keyword");
4    }
5    this.firstName = firstName;
6    this.lastName = lastName;
7    this.gender = gender;
8  }
```

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

```
1  function Person(name) {
2    this.name = name;
3  }
4  var p = Person("John");
5  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

```
1  var Person = function(firstName, lastName) {
2        this.firstName= firstName;
3        this.lastName = lastName;
4        fullName: function() {
5        return this.firstName + " " + this.lastName;
6     }
7  }
8  var person = new Person("John", "Doe");
9  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

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 ?

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

- What will happen if we do the following instead

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

Prototype

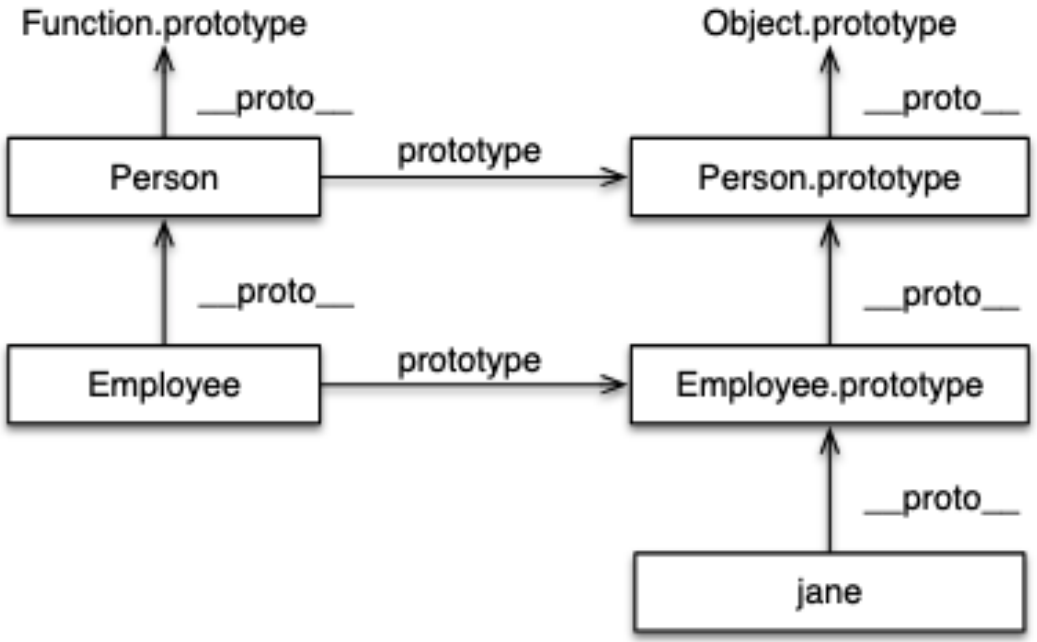
- what is the prototype value of a “Person” object ?

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

- What will happen if we do the following instead

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

Prototype Inheritance



Prototype Example

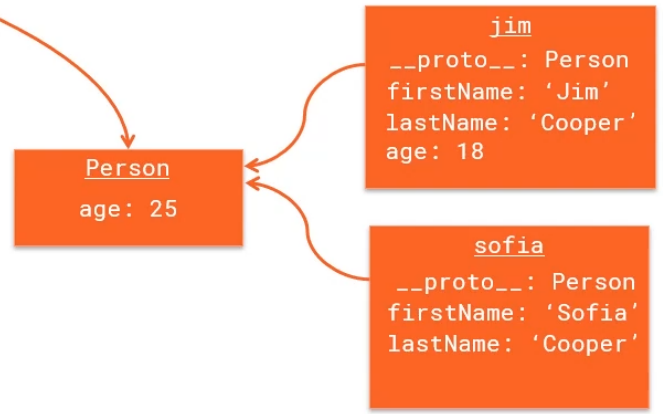

```
1  function Person(firstName, lastName) {
2    this.firstName = firstName;
3    this.lastName = lastName;
4  }
5
6  Person.prototype.age = 29;
7  let jim = new Person('Jim', 'Cooper');
8  let sofia = new Person('Sofia', 'Cooper');
9
10 jim.age = 18;
11 console.log(jim.age); // ?
12
13 Person.prototype.age = 25;
14
15 console.log(jim.age);   // ?
16 console.log(sofia.age); // ?
```

Prototype Example


```
1  function Person(firstName, lastName) {
2    this.firstName = firstName;
3    this.lastName = lastName;
4  }
5
6  Person.prototype.age = 29;
7  let jim = new Person('Jim', 'Cooper');
8  let sofia = new Person('Sofia', 'Cooper');
9
10 jim.age = 18;
11 console.log(jim.age); // 18
12
13 Person.prototype.age = 25;
14
15 console.log(jim.age);   // 18
16 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


```
1  var p1 = new Person("John", "Smith", "Male");
2
3  Person.prototype.print = function() {
4     console.log( "Person: " + this.firstName
5           + this.lastName + this.gender + "\n");
6  }
7
8  var p2 = new Person("Linda", "James", "Female");
9  p1.print();
10 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

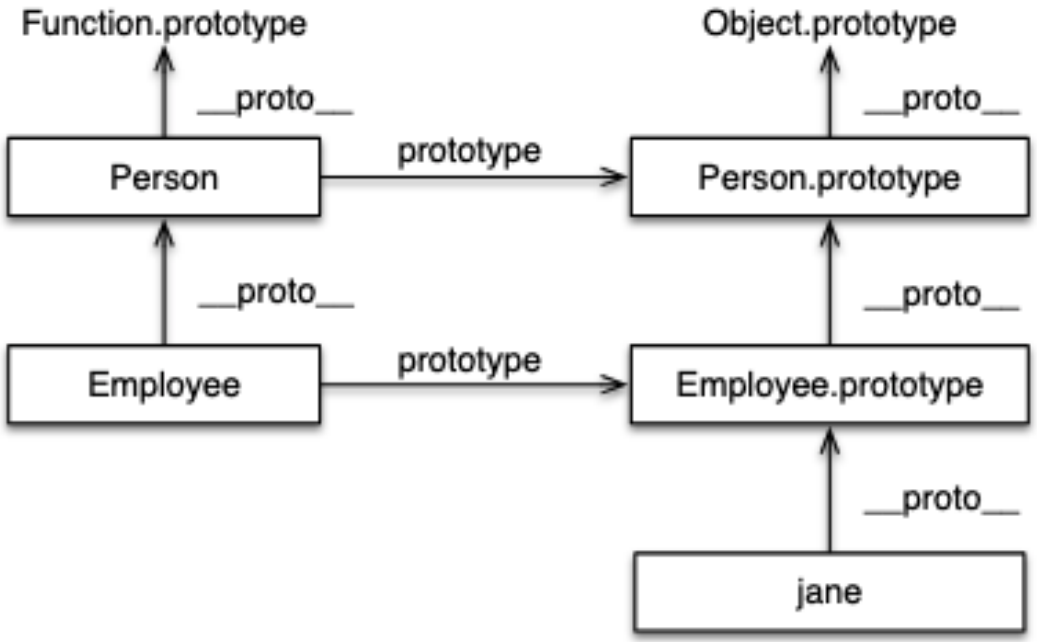
Person has:

- firstname, lastName, gender

Solution: Implement Employee


```
1  function Person(firstName, lastName, gender) {
2    this.firstName = firstName;
3    this.lastName = lastName;
4    this.gender = gender;
5  }
6
7  var Employee = function(firstName, lastName, gender, title)
       {
8    Person.call( this, firstName, lastName, gender );
9    this.title = title;
10 }
11
12 Employee.prototype = new Person();
13   /* Why should you create a new person object ? */
14
15 Employee.prototype.constructor = Employee;
16
17 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