CSM
breaking thought barriers

Angular Training

Session -6

Outlines

CSM
breaking thought barriers

- let , var ,const
- Variable Scopes
- Hoisting
- Functions
- Objects & Prototypes
- Classes
- Promises & Async/Await
- Iterators & Generators
- Modules
- Error handling
- JavaScript DOM
- JavaScript Runtime

Variable scope

- Scope determines the visibility and accessibility of a variable.

The global scope

Local scope

Block scope (started from ES6)

Global scope

- When the JavaScript engine executes a script, it creates a global execution context.

- It also assigns variables that you declare outside of functions to the global execution context. These variables are in the global scope. They are also known as global variables

Local scope

- The variables that you declare inside a function are local to the function. They are called local variables.
- When the JavaScript engine executes the say() function, it creates a function execution context.

Block scope

- ES6 provides the let and const keywords that allow you to declare variables in block scope.

- Generally, whenever you see curly brackets { }, it is a block. It can be the area within the if, else, switch conditions or for, do while, and while loops.

Global variable leaks: the weird part of JavaScript

```
function getCounter() {
counter = 10;
return counter;
}
console.log(getCounter());
```

- Earlier before 2015, the only keyword available to declare variables was the var keyword.
- ES6 -> let ,const

Scope of var, let and const

var : Function in which the variable is declared

let: Block in which the variable is declared

const: Block in which the variable is declared

- let is similar to var. It allows us to declare variables in our local scope.
- The differences are that let is:
  - ✓ not hoisted
  - ✓ block-scoped

Rules of Thumb:

- ✓ Don’t use var, because let and const is more specific
- ✓ Default to const, because it cannot be re-assigned or re-declared
- ✓ Use let when you want to re-assign the variable in future
- ✓ Always prefer using let over var and const over let

Unchanging Values With const

Example 1

```
const taxRate = 0.1;
const total = 100 + (100 * taxRate);
// Skip 100 lines of code
console.log(`Your Order is ${total}`);
```

```
var taxRate = 0.1;
var total = 100 + (100 * taxRate);
// Skip 100 lines of code
console.log(`Your Order is ${total}`);
```

Example 2

```
const taxRate = 0.1;
const shipping = 5.00;
let total = 100 + (100 * taxRate) + shipping;
// Skip 100 lines of code
console.log(`Your Order is ${total}`);
```

Changing the value of a const variable

```
const discountable = [];
const cart = [
    {
        item: 'Book',
        discountAvailable: false,
    },
    {
        item: 'Magazine',
        discountAvailable: true,
    },
];
// Skip some lines
for (let i = 0; i < cart.length; i++) {
    if (cart[i].discountAvailable) {
        discountable.push(cart[i]);
    }
}
console.log(discountable);
```

What is Javascript hoisting?

- **Hoisting** is Javascript's default behavior of moving all declarations to the top of their functional scope.

- **Hoisting variables**

  JS *hoists* the declaration of a variable up and allows it to be used. This way, a variable can be declared after it has been used.

A typical programming case

```
var x;
x = 5;
```

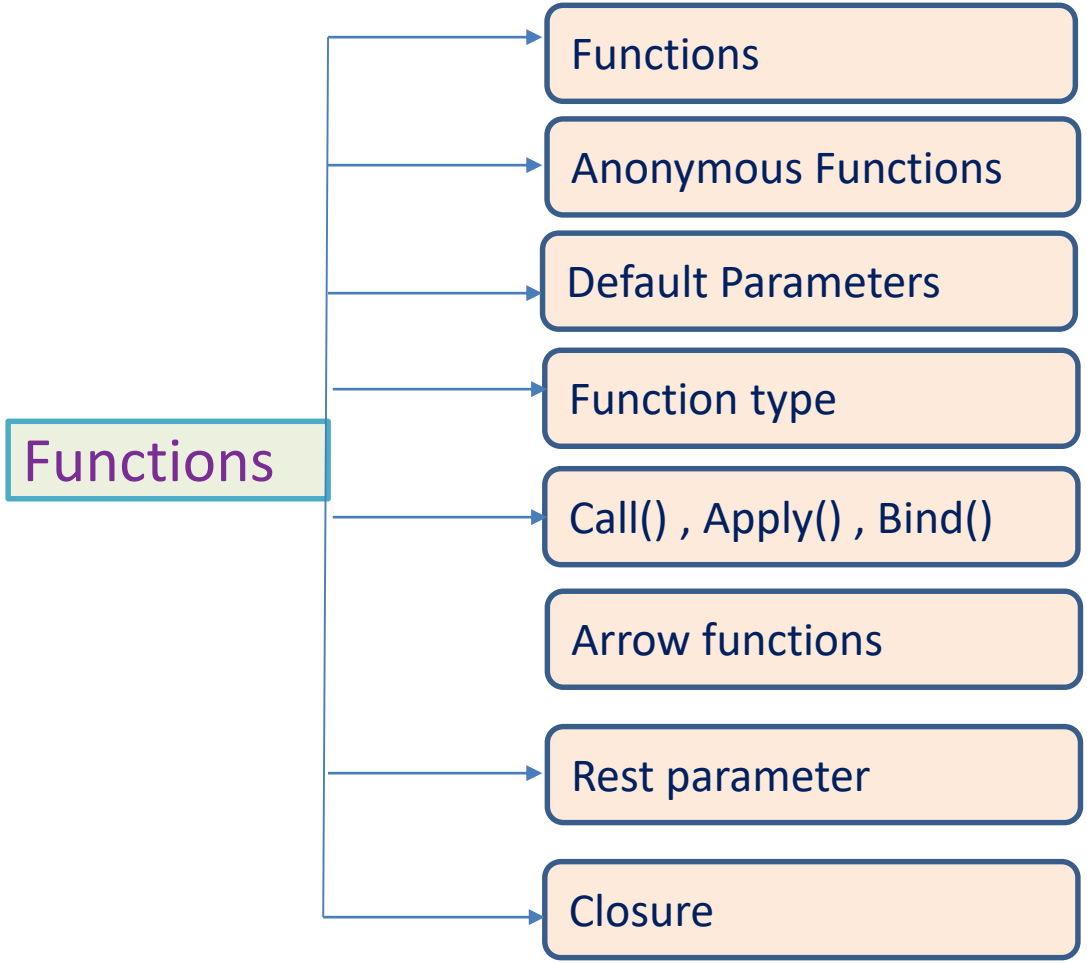
Variable hoisting in JS

```
JS
x = 5;
var x;
```

- **Hoisting functions**

```
func();

function func() {
    var a = 1;
    var b = 2;
    var c = 3;
    console.log(a + " " + b + " " + c);
}
```


Functions
Functions
Anonymous Functions
Default Parameters
Function type
Call() , Apply() , Bind()
Arrow functions
Rest parameter
Closure

Function in JS

1. Declare a function
2. Calling a function
3. Parameters vs. Arguments
4. Returning a value
5. The arguments object
7. Function hoisting
8. Storing functions in variables
9. Passing a function to another function
10. Returning functions from functions

Anonymous Functions

- An anonymous function is a function without a name
- **Note :** If you don't place the anonymous function inside the ()
- An anonymous function is not accessible after its initial creation. Therefore, you often need to assign it to a variable.
- Using anonymous functions as arguments

```
setTimeout (function() {
    console.log('Execute later after 1 second')
}, 1000);
```

```
let taxiing = car.start.bind(aircraft);
taxiing();

car.start.call(aircraft);
```

- The bind() method creates a new function that you can execute later while the call() method executes the function immediately.

- The bind(), call(), and apply() methods are also known as borrowing functions.

Default Parameters

Arrow Function in JS

- An arrow function is a feature introduced in ECMAScript 6 (ES6) for writing shorter and more concise function expressions in JavaScript.
- Arrow functions provide a more streamlined syntax compared to traditional function expressions and also have some differences in how they handle the **this** keyword.

```
const functionName = (parameters) => {

    return result;

};
```

Characteristics Arrow Function

- **Shorter Syntax:**
  Arrow functions are often shorter and more concise compared to traditional function expressions, especially for simple functions.

- **Implicit Return:**
  If the function body consists of a single expression, you can omit the curly braces {} and the return keyword. The result of the expression will be automatically returned.

- **No Binding of this:**
  Arrow functions do not have their own **this** value. Instead, they inherit the **this** value from the enclosing context. This makes them useful when you want to maintain the context of the surrounding code.

- Arrow functions are useful for certain scenarios, they might not be suitable for all cases.

- For example,

  - ✓ arrow functions cannot be used as constructor functions (i.e., they cannot be used with the new keyword)

  - ✓ do not have their own arguments object.

this Keyword

- The this keyword is a special identifier that refers to the current execution context, specifically the object that "owns" the currently executing code.

- The behavior of the this keyword depends on how and where it is used within a function or method.

The behavior of this :

1. Global Context:

In the global context (outside of any function or object), this refers to the global object, which is often the window object in browsers or the global context in Node.js.

2. Function Context:

- In a regular function (not an arrow function), the value of this depends on how the function is called.
- If the function is called as a method of an object, this refers to the object itself.

- If the function is called standalone, this might refer to the global object (in non-strict mode) or be undefined (in strict mode).

The behavior of this :

3. Arrow Function Context :

In an arrow function, this retains the value of this from the enclosing lexical context (the context where the arrow function is defined). It does not have its own binding for this.

4. Method Context:

When a function is a method of an object, this refers to the object itself.

5. Explicitly Setting this:

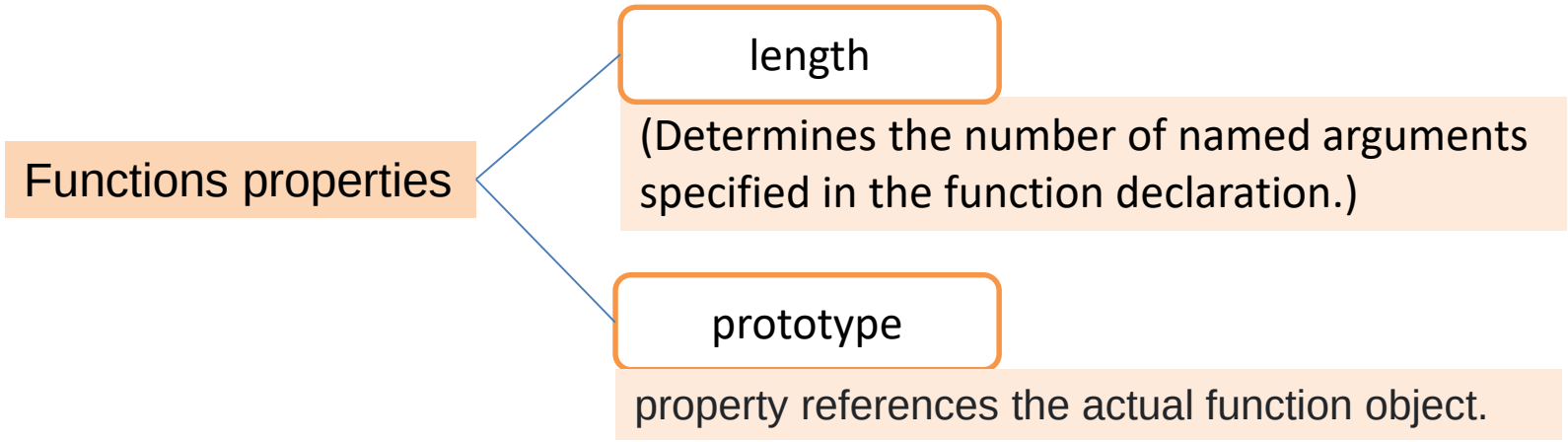
You can explicitly set the value of this using functions like bind, call, and apply.

NOTE :

this can sometimes be complex, especially in nested functions or callback scenarios.

JavaScript Function Type

- In JavaScript, all functions are objects.
- They are the instances of the Function type Because functions are objects, they have properties and methods like other objects.



Function methods: apply, call, and bind

- A function object has three important methods: **apply(), call() and bind().**

- The **apply()** and **call()** methods call a function with a given **this** value and arguments.

- The difference between the apply() and call() is that you need to pass the arguments to the apply() method as an array-like object, whereas you pass the arguments to the call() function individually.

Function methods: apply, call,bind

```
let cat = { type: 'Cat', sound: 'Meow' };
let dog = { type: 'Dog', sound: 'Woof' };

const say = function (message) {
  console.log(message);
  console.log( this.type + ' says ' + this.sound);
};
say();
say.apply(cat, ['What does a cat say?']);
say.apply(dog, ['What does a dog say?']);

say.call(cat, 'What does a cat say?');
say.call(dog, 'What does a dog say?');
```

Bind() Method

- In JavaScript, the bind() method is used to create a new function that has a specific **this** value, and optionally, arguments that are "bound" to it.

- This is particularly useful when you want to explicitly set the value of this within a function, regardless of how the function is called.

- The bind() method does not invoke the function immediately but returns a new function with the specified context and arguments.

```
functionToBind.bind( thisValue[, arg1[, arg2[, ...]]])
```

apply() Method

- In JavaScript, the apply() method is a function method that allows you to invoke a function with a specific this value and an array (or an array-like object) of arguments.

- It is similar to the call() method, but instead of passing individual arguments directly, you pass them as an array or array-like object.

- The apply() method can be useful when you have a function that accepts a variable number of arguments and those arguments are stored in an array. It's also often used to invoke functions with a specific this context and arguments extracted from an array or array-like object.

JavaScript Closures

- A closure is a function that references variables in the outer scope from its inner scope.

- The closure preserves the outer scope inside its inner scope.

Lexical scoping

```
let name = 'John';

function greeting() {
    let message = 'Hi';
    console.log(message + ' '+ name);
}
```

According to lexical scoping, the scopes can be nested and the inner function can access the variables declared in its outer scope.

Lexical scoping

```
function greeting() {
    let message = 'Hi';
    function sayHi() {
        console.log(message);
    }
    sayHi();
}
greeting();
```

JavaScript closures

```
function greeting() {

let message = 'Hi';

function sayHi()
{
   console.log(message);
}
return sayHi;
}
let hi = greeting();
hi();
```

```
function greeting(message)
{
    return function(name){
        return message + ' ' + name;
    }
}
let sayHi = greeting('Hi');
let sayHello = greeting('Hello');
console.log(sayHi('John'));
console.log(sayHello('John'));
```

JavaScript closures in a loop

```
for (var index = 1; index <= 3; index++)
{
  setTimeout(function (){
      console.log('after ' + index + ' second(s):' + index);
    },index * 1000);
}
```

To fix this issue, you need to create a new closure scope in each iteration of the loop.

1) Using the IIFE solution :

We use an immediately invoked function expression (IIFE) because an IIFE creates a new scope by declaring a function and immediately execute it.

```
for (var index = 1; index <= 3; index++)
{
  (function (index)
  {
    setTimeout(function (){
    console.log('after ' + index + ' second(s):' + index);
    }, index * 1000);
  })(index);
}
```

To fix this issue, you need to create a new closure scope in each iteration of the loop.

2) Using let keyword in ES6

```
for (let index = 1; index <= 3; index++) {
    setTimeout(function () {
        console.log('after ' + index + ' second(s):' + index);
    }, index * 1000);
}
```

```
function createCounter() {
  let count = 0;
  function increment() {
    count++;
  }
  function getCount() {
    return count;
  }
  return {
    increment,
    getCount
  };
}
const counter = createCounter();
console.log( counter.getCount() ); // Output: 0
counter.increment();
counter.increment();
console.log( counter.getCount()); // Output: 2
```

Case studies where closures are commonly used in JavaScript :

1. Event Handlers in UI Development:

- Event handlers often need to access variables from their containing scope.
- For example, in an event handler, you might need to access data related to a specific UI element, and closures help you encapsulate and retain that data.

```
function createButton() {
  let count = 0;
  const button = document.createElement('button');
  button.textContent = 'Click me';
  button.addEventListener('click', function() {
    count++;
    console.log(`Button clicked ${count} times.`);
  });
  return button;
}
const buttonElement = createButton();
document.body.appendChild(buttonElement);
```

2. Module Pattern and Private Members:

Closures are often used to create modules in JavaScript, allowing you to define private and public members. This pattern helps in achieving encapsulation and information hiding.

```
const calculator = (function() {
  let result = 0;
  function add(a, b) {
    result = a + b;
  }
  function getResult() {
    return result;
  }
  return {
    add,
    getResult
  };
})();
calculator.add(5, 3);
console.log( calculator.getResult()); // Output: 8
```

3. Caching and Memoization:

Closures are useful for implementing caching and memoization techniques. You can use closures to store previously computed values and avoid redundant calculations.

```
function memoize(func) {
        const cache = {};
        return function(...args) {
        const key = JSON.stringify (args);
        if (cache[key] === undefined)
        {
         cache[key] = func(...args);
        } return cache[key]; };
     }
    function expensiveOperation (x, y) {
    console.log('Performing expensive operation...');
    return x + y;
    }
const memoizedOperation = memoize(expensiveOperation);
console.log(memoizedOperation(2, 3));
console.log(memoizedOperation(2, 3));
```

4. Callbacks and Asynchronous Operations:

- Closures are commonly used with asynchronous operations, such as callbacks, to capture and maintain the context of the surrounding code even after the asynchronous operation completes.

```
function fetchData(url, callback) {
  // Simulate an asynchronous operation (e.g., an HTTP
request)
  setTimeout(function () {
    var data = "Data from " + url;
    callback(data);
  }, 1000);
}

function processData(data) {
  console.log("Processing data:", data);
}

fetchData("https://example.com/api/data", processData);
```

```
function fetchData( url , callback) {
    fetch(url)
    .then(response => response.json())
    .then(data => callback(data))
    .catch(error => console.error(error));
}

const apiUrl = 'https://fakestoreapi.com/products/1';
fetchData(apiUrl, function(data) {
  console.log('Fetched data:', data);
});
```

Objects & Prototypes :

object methods :

- ✓ An object is a collection of key/value pairs or properties.
- ✓ When the value is a function, the property becomes a method. We use methods to describe the object behaviors.

```
let person = {
 firstName: 'John',
 lastName: 'Doe’
};
```

The this value:

Constructor Function :

```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};
```

- ✓ We often need to create many similar objects like the person object.
- ✓ To do that, you can use a constructor function to define a **custom type** and the new operator to create multiple objects from this type.

A constructor function is a regular function with the following convention:

- ✓ The name of a constructor function starts with a capital letter
- ✓ A constructor function should be called only with the new operator.

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

```
let person1 = new Person('Jane','Doe')
let person2 = new Person('James','Smith')
```

```
function Person(firstName, lastName)
{
        // this = {};
        // add properties to this
        this.firstName = firstName;
        this.lastName = lastName;
        // return this;
}
```

Adding methods to JavaScript constructor functions

```
function Person(firstName, lastName)
{
    this.firstName = firstName;
    this.lastName = lastName;
    this.getFullName = function ()
    {
        return this.firstName + " " + this.lastName;
    };
}
```

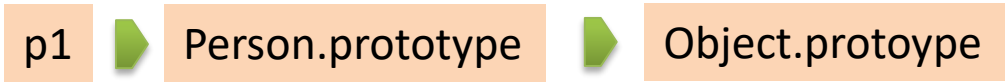
The problem with the constructor function is that when you create multiple instances of the Person, the this.getFullName() is duplicated in every instance, which is **not memory efficient**.

To resolve this, you can use the prototype so that all instances of a custom type can share the same methods.

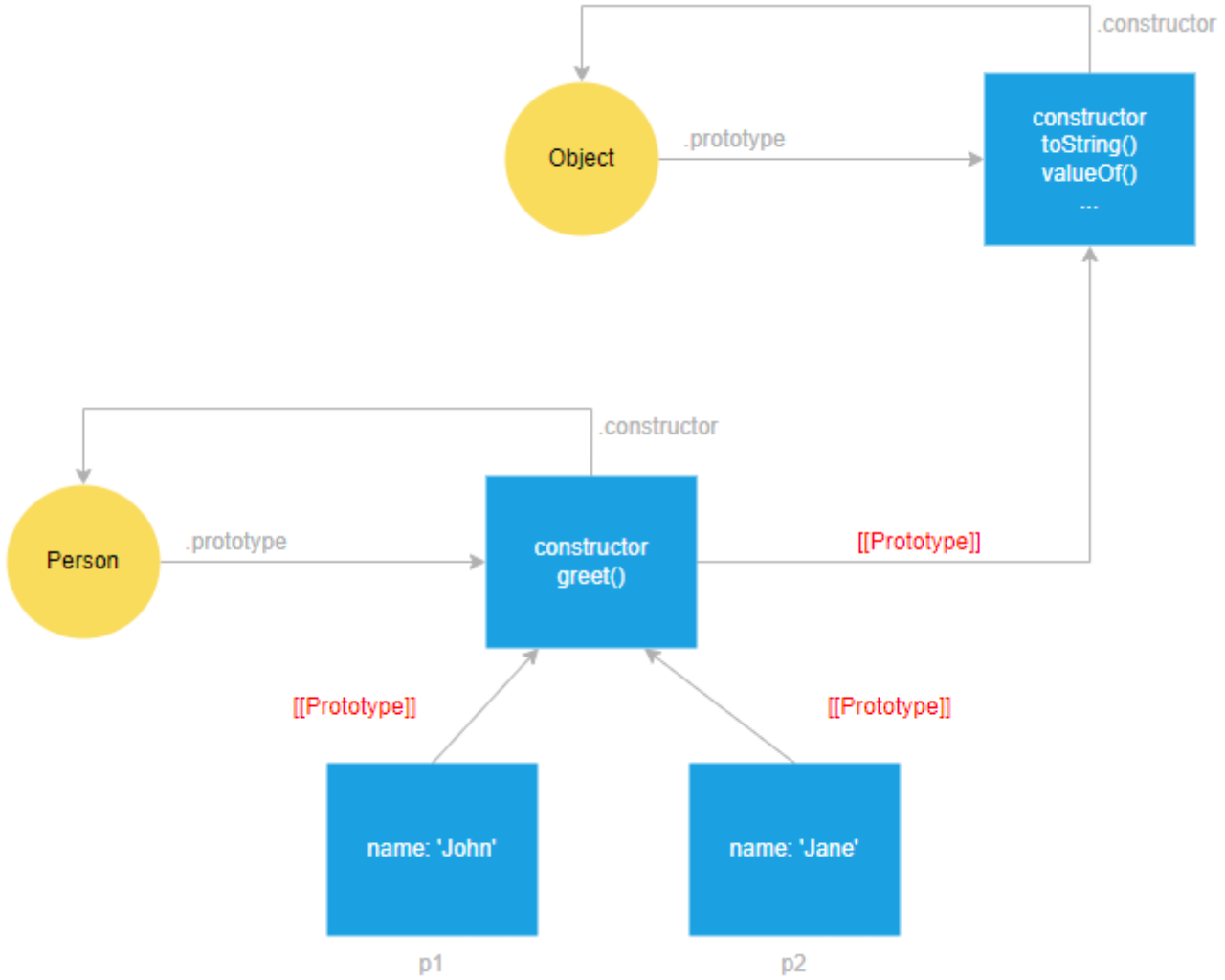
Defining methods in the JavaScript prototype object

```
Person.prototype.greet = function() {
        return "Hi, I'm " + this.name + "!";
}
```

```
let p1 = new Person('John');
```

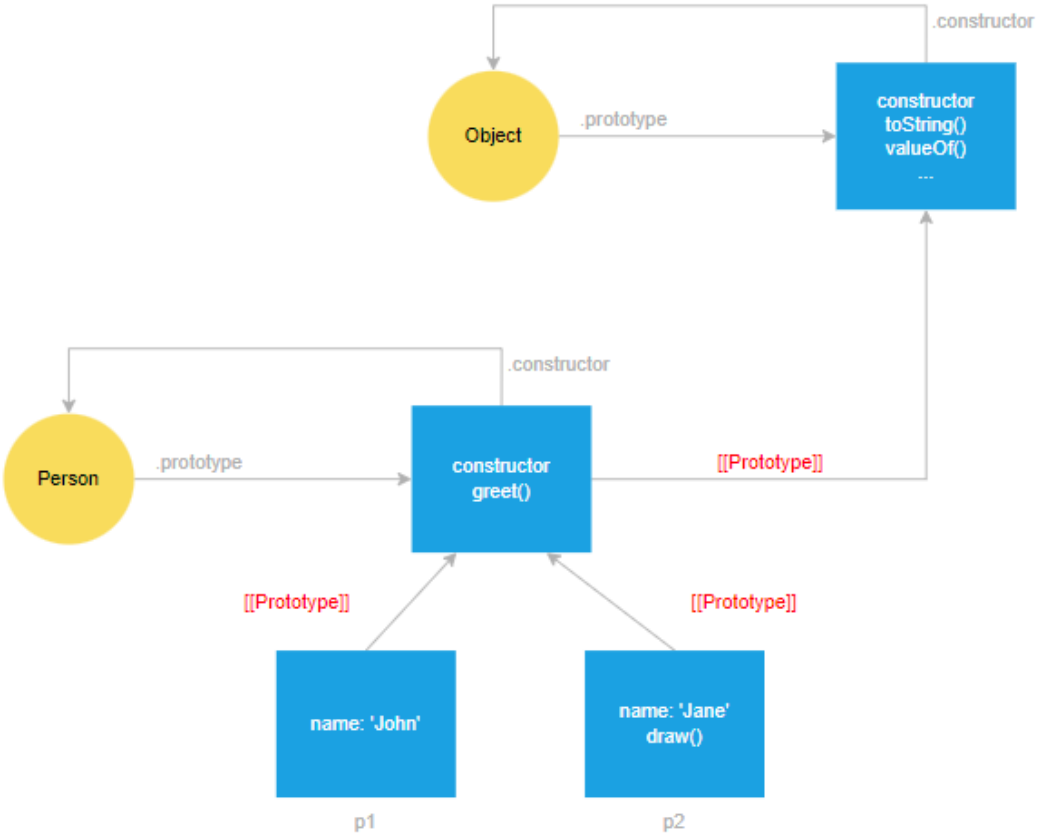


(Prototype Chain)


.constructor
Object
.prototype
constructor
toString()
valueOf()
...
.constructor
Person
.prototype
constructor
greet()
[[Prototype]]
[[Prototype]]
[[Prototype]]
name: 'John'
name: 'Jane'
p1
p2

Defining methods in an individual object

```
p2.draw = function () {
    return "I can draw.";
};
```



Returning from constructor functions

- ✓ A constructor function implicitly returns this that set to the newly created object.

If it has a return statement, then here's are the rules:

1. If return is called with an object, the constructor function returns that object instead of this.

2. If return is called with a value other than an object, it is ignored.

Calling a constructor function without the new keyword

```
let person = Person('John', 'Doe');
```

- ✓ The this inside the Person function doesn't bind to the person variable but the global object.

- ✓ To prevent a constructor function to be invoked without the new keyword, ES6 introduced the new.target property.

- ✓ If a constructor function is called with the new keyword, the new.target returns a reference of the function. Otherwise, it returns undefined.

```
if (!new.target) {
      throw Error("Cannot be called without the new
keyword");
   }
```

```
if (!new.target) {
       return new Person(firstName, lastName);
   }
```

JavaScript Prototype

- ✓ In JavaScript, objects can inherit features from one another via prototypes.
- ✓ Every object has its own property called prototype. Because a prototype itself is also another object, the prototype has its own prototype. This creates a something called **prototype chain**.
- ✓ The prototype chain ends when a prototype has null for its own prototype.
- ✓ When you access a property of an object, if the object has that property, it'll return the property value.
- ✓ However, if you access a property that doesn't exist in an object, the JavaScript engine will search in the prototype of the object.
- ✓ If the JavaScript engine cannot find the property in the object's prototype, it'll search in the prototype's prototype until either it finds the property or reaches the end of the prototype chain.

Getting prototype linkage

The __proto__ is pronounced as under proto.

The __proto__ is an accessor property of the Object.prototype object.

It exposes the internal prototype linkage ([[Prototype]]) of an object through which it is accessed.

The __proto__ has been standardized in ES6 to ensure compatibility for web browsers. However, it may be deprecated in favor of Object.getPrototypeOf() in the future. Therefore, you should never use the __proto__ in your production code.

The p1.__proto__ exposes the [[Prototype]] that references the Person.prototype object.

Thank You