

Javascript OOPS concepts

In JavaScript, the three main ways to create objects and define class-like behavior are **constructor functions**, **ES6 classes**, and **Object.create()**. Each has its own approach to object creation and inheritance, and understanding the differences between them can help you decide when to use each method. But ES6 Classes are using nowadays as their Modern and Standard .

Constructor function

A constructor function in JavaScript is a function used to create and initialize objects. When you use the **new** keyword with a constructor function, it helps create a new instance of an object based on that constructor.

Key Steps in the Process:

1. **New Object ({}):** When you use the **new** keyword, it creates a new empty object.
2. **Function Call:** The constructor function is called, and **this** inside the constructor refers to the newly created object.
3. **Link to Prototype:** The new object's prototype is set to the constructor's **prototype** property, which is where you can define methods that should be shared by all instances.
4. **Return Object:** If the constructor does not explicitly return an object, it returns the newly created object by default.

Rules and best practices for creating a constructor function in JavaScript:

i) Function Name Capitalization

- By convention, **the name of a constructor function should start with an uppercase letter**. This helps distinguish constructor functions from regular functions (which typically start with lowercase letters).

Function Declaration (Traditional Way)

```
function Person(name, age) {  
  this.name = name;  
  this.age = age;  
}
```

Function Expression (Anonymous Function)

```
const Person = function(name, age) {  
  this.name = name;  
  this.age = age;  
};
```

ii) Use the **new** Keyword

A constructor function should always be invoked with the **new** keyword. If the **new** keyword is not used, the function will not work as expected, and **this** will not refer to a new object.

When using **new**, the following happens:

1. A new empty object (`{}`) is created.
2. The constructor function is called with the new object as **this**.
3. The new object is returned from the function (implicitly).

Example:

```
const person = new Person("John", 30); // Correct usage
```

iii) The **this** Keyword

Inside a constructor function, **this** refers to the newly created object. You can add properties to this object using **this.propertyName = value**;

Example:

```
function Animal(name, sound) {  
  this.name = name;  
  this.sound = sound;  
}
```

iv) **Avoiding Arrow Functions as Constructors**

- Arrow functions should not be used as constructor functions because they do not have their own **this** context. Arrow functions inherit **this** from the enclosing scope, which leads to unexpected results when using **new**.

Example:

```
// Incorrect usage: arrow functions cannot be used as constructors
```

```
const Person = (name, age) => {  
  
  this.name = name;  
  
  this.age = age;  
  
};
```

```
const person1 = new Person("Alice", 25); // Error
```

Ex

```
//constructor function and new operator
const Employee = function (empName, salary) {
  this.empName = empName;
  this.salary = salary;
};

const emp1 = new Employee('ram', 2000);
console.log(emp1); //Employee {empName: 'ram', salary: 2000}
// 1. New {} is created
//2. function is called , this={}
//3. {} linked to prototype
//4. function automatically return {}
const emp2 = new Employee('dhruv', 5000);
console.log(emp2); //Employee {empName: 'dhruv', salary: 5000}
const emp3 = new Employee('kl', 12000);
console.log(emp3); //Employee {empName: 'kl', salary: 12000}
console.log(emp1 instanceof Employee); //true
console.log(emp2 instanceof Employee); //true
console.log(emp3 instanceof Employee); //true
```

Prototypes

In JavaScript, **prototypes** are an essential feature of the language's object-oriented behavior. They allow objects to inherit properties and methods from other objects, enabling a form of inheritance.

All JavaScript objects inherit properties and methods from a prototype.

- **Date** objects inherit from **Date.prototype**
- **Array** objects inherit from **Array.prototype**
- **Person** objects inherit from **Person.prototype**

The **Object.prototype** is on the top of the prototype inheritance chain:

Date objects, Array objects, and Person objects inherit from Object.prototype.

Object.prototype: The root of all JavaScript objects is Object.prototype, which has a set of default methods, such as toString(), hasOwnProperty(), etc. If a property is not found in an object's prototype chain, JavaScript checks Object.prototype.

Now, we will add a method calculateAge() to the Prototype property in a Person function constructor which will be inherited by the different objects. Below is the code for this:-

```
// Function constructor

function Person(name, job, yearOfBirth) {

    this.name = name;

    this.job = job;

    this.yearOfBirth = yearOfBirth;

}

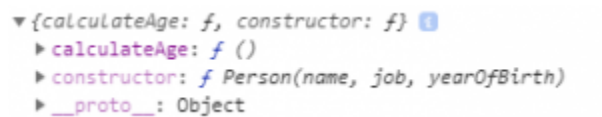
Person.prototype.calculateAge = function () {

    console.log('The current age is: ' + (2019 - this.yearOfBirth));

}

console.log(Person.prototype);
```

Output



```
▼ {calculateAge: f, constructor: f} ⓘ
  ► calculateAge: f ()
  ► constructor: f Person(name, job, yearOfBirth)
  ► __proto__: Object
```

In the above image, we can see the calculateAge() method gets added to the Prototype property. Now, we will create 2 different objects which will inherit the calculateAge() method and remember, When a certain method(or property) is called, it first checks inside the object but when it isn't found, then search moves on Object's prototype.

```
// Function constructor

function Person(name, job, yearOfBirth) {

    this.name = name;

    this.job = job;

    this.yearOfBirth = yearOfBirth;
}

// Adding calculateAge() method to the Prototype property

Person.prototype.calculateAge = function () {

    console.log('The current age is: ' + (2019 - this.yearOfBirth));

}

console.log(Person.prototype);

// Creating Object Person1

let Person1 = new Person('Jenni', 'clerk', 1986);

console.log(Person1)

let Person2 = new Person('Madhu', 'Developer', 1997);

console.log(Person2)

Person1.calculateAge();

Person2.calculateAge();
```

Output

```

  ▶ {calculateAge: f, constructor: f}
  ▶ Person {name: "Jenni", job: "clerk", yearOfBirth: 1986}
  ▶ Person {name: "Madhu", job: "Developer", yearOfBirth: 1997}
  The current age is: 33
  The current age is: 22

```

Here, we created two Objects Person1 and Person2 using constructor function Person, when we called Person1.calculateAge() and Person2.calculateAge(), First it will check whether it is present inside Person1 and Person2 object, if it is not present, it will move Person's Prototype object and prints the current age, which shows **Prototype property enables other objects to inherit all the properties and methods of function constructor**.

The `__proto__` property is a way to access an object's prototype directly the `__proto__` property allows you to access an object's prototype directly. It provides a way to inspect or modify the prototype chain of an object. However, it's generally recommended to use the modern `Object.getPrototypeOf()` and `Object.setPrototypeOf()` methods instead of `__proto__`.

```

'use strict';
//constructor function and new operator
const Employee = function (empName, salary) {
  this.empName = empName;
  this.salary = salary;
};

const emp1 = new Employee('ram', 2000);
console.log(emp1); //Employee {empName: 'ram', salary: 2000}
// 1. New {} is created
//2. function is called , this={}
//3. {} linked to prototype
//4. function automatically return {}
const emp2 = new Employee('dhruv', 5000);
console.log(emp2); //Employee {empName: 'dhruv', salary: 5000}
const emp3 = new Employee('kl', 12000);
console.log(emp3); //Employee {empName: 'kl', salary: 12000}

console.log(emp1 instanceof Employee); //true
console.log(emp2 instanceof Employee); //true

```

```
console.log(emp3 instanceof Employee); //true

Employee.prototype.fullDetails = function () {
  console.log('Name:' + this.empName, 'salary: ' + this.salary);
};

console.log(Employee.prototype);
emp1.fullDetails(); //Name:ram salary: 2000
emp2.fullDetails(); //Name:dhruv salary: 5000

//we can add property using prototype
Employee.prototype.age = 23;
console.log(Employee.prototype);
// Accessing the prototype of emp1
console.log(emp1.__proto__); // Logs Employee.prototype

// Verifying that Employee.prototype is the prototype of emp1
console.log(emp1.__proto__ === Employee.prototype); // true

// Accessing the prototype using Object.getPrototypeOf
console.log(Object.getPrototypeOf(emp1) === Employee.prototype); // true

console.log(Employee.prototype.isPrototypeOf(emp1)); // true
console.log(Employee.prototype.isPrototypeOf(emp2)); // true
// Accessing the fullDetails method via __proto__
console.log(emp1.__proto__.fullDetails);
// Logs the function definition of fullDetails
/*
f () {
  console.log('Name:' + this.empName, 'salary: ' + this.salary);
}
*/
console.log(emp1.hasOwnProperty('empName')); //true
console.log(emp1.hasOwnProperty('salary')); //true
```

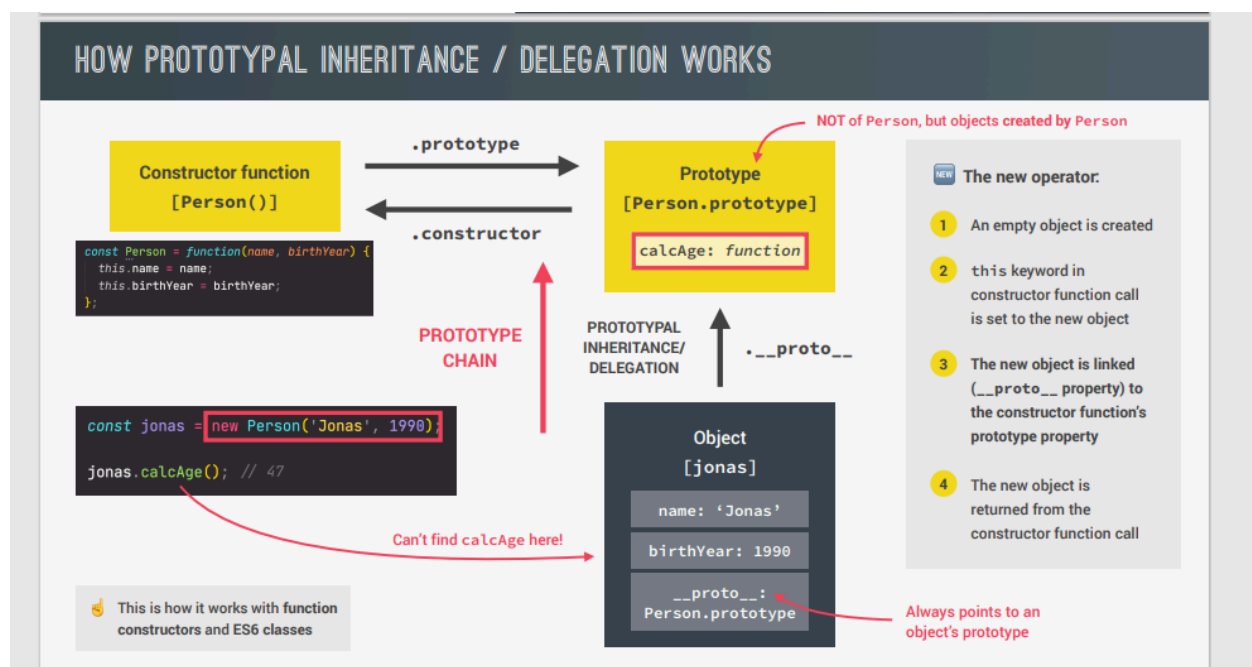
```
console.log(empl1.hasOwnProperty('age')); //false
```

In JavaScript, **prototype inheritance** and the **prototype chain** are key concepts in object-oriented programming. They enable objects to inherit properties and methods from other objects.

Prototype Inheritance

Prototype inheritance allows an object to use the properties and methods of another object. In JavaScript:

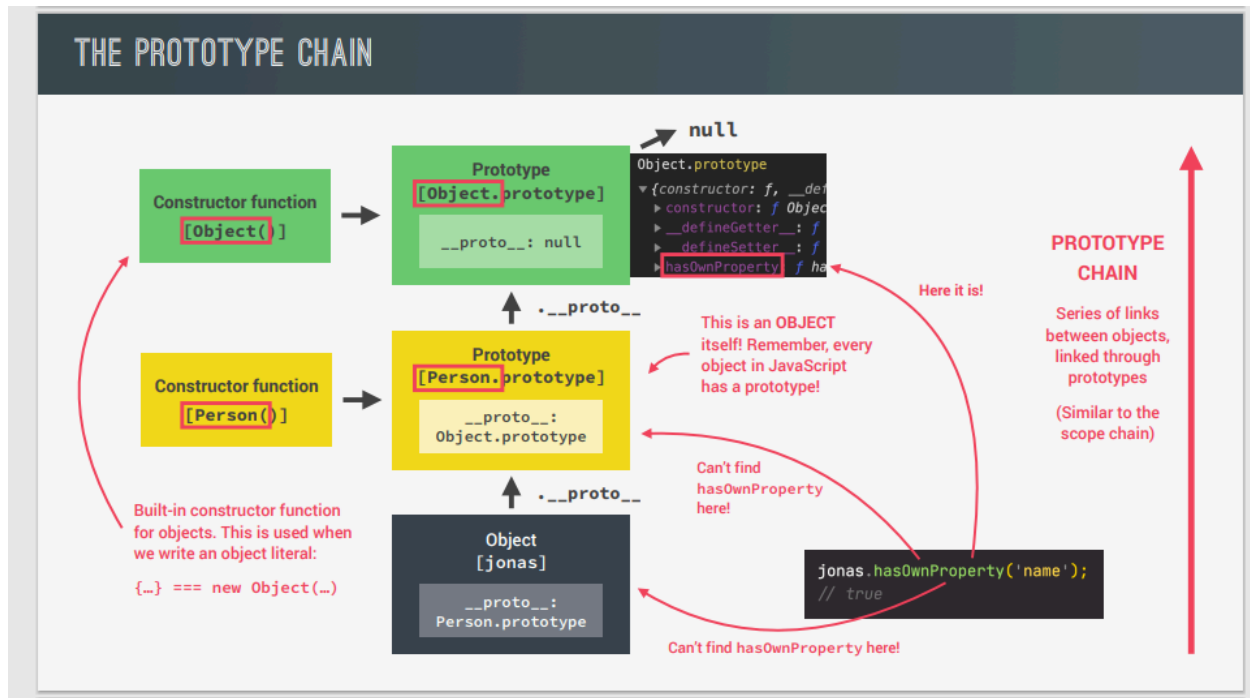
1. Every object has a hidden, internal property called `[[Prototype]]`, which can be accessed using `Object.getPrototypeOf(obj)` or the `__proto__` property (deprecated, but still widely used).
2. When a property or method is accessed on an object, JavaScript checks the object itself first. If it doesn't find the property, it looks up the chain via the `[[Prototype]]`.



Prototype Chain

The **prototype chain** is the mechanism that links objects together via their `[[Prototype]]`.

- When you try to access a property or method, JavaScript looks for it in the object itself. If it doesn't find it, it looks in the object's prototype.
- This lookup continues up the chain until it reaches **null** (the end of the chain).



prototype inheritance and the prototype chain on built-in objects like array

```
'use strict';
//constructor function and new operator
const Employee = function (empName, salary) {
  this.empName = empName;
  this.salary = salary;
};

const emp1 = new Employee('ram', 2000);
console.log(emp1); //Employee {empName: 'ram', salary: 2000}

// console.log(emp3 instanceof Employee); //true

Employee.prototype.fullDetails = function () {
  console.log('Name:' + this.empName, 'salary: ' + this.salary);
};
```

```

};

console.log(emp1.__proto__); //parent prototype
console.log(emp1.__proto__.__proto__); //Object prototype(parent)
console.log(emp1.__proto__.__proto__.__proto__); //null , it has reached highest Object
prototype chain

//arrays

const arr = [5, 6, 7, 8, 9]; //new Array()
console.log(arr.__proto__);
console.log(arr.__proto__.__proto__);
console.log(arr.__proto__ === Array.prototype); //true

```

```

▶ Employee {empName: 'ram', salary: 2000} script.js:9
▼ {fullDetails: f} i script.js:57
  ▶ fullDetails: f ()
  ▶ constructor: f (empName, salary)
  ▶ [[Prototype]]: Object
  {__defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__:
  f, __lookupSetter__: f, ...} script.js:58
null script.js:59
▶ [at: f, concat: f, copyWithin: f, fill: f, find: f, ...] script.js:64
  {__defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__:
  f, __lookupSetter__: f, ...} script.js:65
true script.js:66

```

Coding Challenge #1

Your tasks:

1. Use a constructor function to implement a 'Car'. A car has a 'make' and a 'speed' property. The 'speed' property is the current speed of the car in km/h
2. Implement an 'accelerate' method that will increase the car's speed by 10, and log the new speed to the console

3. Implement a 'brake' method that will decrease the car's speed by 5, and log the new speed to the console
4. Create 2 'Car' objects and experiment with calling 'accelerate' and 'brake' multiple times on each of them

Test data:

Data car 1: 'BMW' going at 120 km/h

Data car 2: 'Mercedes' going at 95 km/h

```
// Constructor function to define a Car
const Car = function (make, speed) {
  // Initialize car properties
  this.make = make; // The brand or make of the car
  this.speed = speed; // The current speed of the car in km/h
};
```

```
/*
```

ES6 classes provide a more modern and readable way to define constructor functions and methods.

The equivalent ES6 class for the Car constructor function would look like this:

```
class Car {
  constructor(make, speed) {
    this.make = make; // The brand or make of the car
    this.speed = speed; // The current speed of the car in km/h
  }
}
```

```
*/
```

```
// Add an 'accelerate' method to the prototype of Car
```

```
Car.prototype.accelerate = function () {
  // Increase the speed of the car by 10
  this.speed += 10;
  // Log the updated speed along with the car's make
  console.log(`${this.make} is going at ${this.speed} km/h`);
};
```

```

// Add a 'brake' method to the prototype of Car
Car.prototype.brake = function () {
  // Decrease the speed of the car by 5
  this.speed -= 5;
  // Log the updated speed along with the car's make
  console.log(`${this.make} is going at ${this.speed} km/h`);
};

// Create a new car instance for BMW with an initial speed of 120 km/h
const bmw = new Car('BMW', 120);
// Log the BMW instance
console.log(bmw); // Output: Car { make: 'BMW', speed: 120 }

// Create a new car instance for Mercedes with an initial speed of 95 km/h
const mercedes = new Car('Mercedes', 95);
// Log the Mercedes instance
console.log(mercedes); // Output: Car { make: 'Mercedes', speed: 95 }

// Call the 'accelerate' method on the BMW instance
bmw.accelerate(); // Output: BMW is going at 130 km/h
// Call the 'brake' method on the BMW instance
bmw.brake(); // Output: BMW is going at 125 km/h

// Call the 'accelerate' method on the Mercedes instance
mercedes.accelerate(); // Output: Mercedes is going at 105 km/h
// Call the 'brake' method on the Mercedes instance
mercedes.brake(); // Output: Mercedes is going at 100 km/h

```

ES6 classes

In JavaScript, **ES** stands for **ECMAScript**, which is the standard specification that JavaScript is based on. ECMAScript defines the rules and guidelines for the language's syntax, features, and behaviors.

ES6 (also known as **ECMAScript 2015**) is the 6th version of ECMAScript and brought significant improvements and new features to JavaScript like **Let and Const** ,**Arrow Functions** ,**Classes** ,**Template Literals** ,**Destructuring Assignment**, **Modules** ,**Promises** , and etc...

Before ES6 (Using Constructor Functions)

In pre-ES6 JavaScript, classes were simulated using constructor functions and prototypes.

In pre-ES6 JavaScript, classes were simulated using constructor functions and prototypes.

// Constructor function

```
function Person(name, age) {
```

```
  this.name = name;
```

```
  this.age = age;
```

```
}
```

```
/*
```

```
const Person = function(name, age) {
```

```
  this.name = name;
```

```
  this.age = age;
```

```
};
```

```
*/
```

// Adding methods using the prototype outside constructor

```
Person.prototype.greet = function () {
```

```
  console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
```

```
};
```

// Creating an instance

```
const person1 = new Person('Alice', 25);
```

```
person1.greet(); // Output: Hello, my name is Alice and I am 25 years old.
```

ES6 Class Approach

ES6 introduced the `class` syntax, which provides a more modern and cleaner way to define classes.

ES6 introduced the `class` syntax, which provides a more modern and cleaner way to define classes.

// Class declaration

```
class Person {  
  
  constructor(name, age) {  
  
    this.name = name;  
  
    this.age = age;  
  
  }  
  
  // Methods are directly placed inside the class body and outside the constructor without a  
  // prototype . Methods will be added to .prototype property internally  
  
  greet() {  
  
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);  
  
  }  
  
}
```

// Creating an instance

```
const person1 = new Person('Alice', 25);  
  
person1.greet(); // Output: Hello, my name is Alice and I am 25 years old.
```

In this version:

- The `constructor` method is automatically called when an instance is created with `new`.
- Methods are directly placed inside the class body.

Comparison:

- **Syntax:** ES6 classes are simpler and cleaner compared to the constructor function approach.
- **Methods:** In the constructor function, methods are added using the prototype, while in the ES6 class syntax, methods are defined directly inside the class body.
- **Inheritance:** ES6 `class` provides easier syntax for inheritance with `extends` and `super()`, making it easier to work with inheritance chains.

Setter and Getter

In JavaScript, **setters** and **getters** are special methods that allow you to define how a property of an object is set or accessed. These methods are typically used to control the access to an object's properties.

Getter

A **getter** method is used to retrieve the value of an object's property. It is defined using the `get` keyword.

Setter

A **setter** method is used to set the value of an object's property. It is defined using the `set` keyword.

Example with Getter and Setter in a Class

```
class Person {  
  constructor(name, age) {  
    this._name = name; // Using an underscore to indicate a private property  
    this._age = age;  
  }  
  
  // Getter for 'name'  
  get name() {  
    return this._name;  
  }  
  
  // Setter for 'name'  
  set name(newName) {
```

```
    if (newName && newName.length > 0) {
        this._name = newName;
    } else {
        console.log("Invalid name!");
    }
}

// Getter for 'age'
get age() {
    return this._age;
}

// Setter for 'age'
set age(newAge) {
    if (newAge >= 0) {
        this._age = newAge;
    } else {
        console.log("Invalid age!");
    }
}

// Creating an instance
const person1 = new Person('Alice', 25);

// Using getter to access 'name' and 'age'
console.log(person1.name); // Output: Alice
console.log(person1.age); // Output: 25

// Using setter to modify 'name' and 'age'
person1.name = 'Bob';
person1.age = 30;

console.log(person1.name); // Output: Bob
console.log(person1.age); // Output: 30

// Invalid setter usage
person1.name = ""; // Output: Invalid name!
person1.age = -5; // Output: Invalid age!
```


How it works:

- The **getter** allows you to access the value of a private property (like `_name` or `_age`) directly, as if you were accessing a regular property.
- The **setter** allows you to modify the private properties. It can include logic to validate the data before it changes the value of the property.

Why Use Getters and Setters?

1. **Encapsulation:** Getters and setters help to control how properties of an object are accessed and modified. You can hide the actual data structure (like `_name` or `_age`) and expose only the required behavior.
2. **Validation:** Setters can be used to validate data before modifying an object's property. For example, you can ensure that a `name` is not empty or that `age` is a positive number.
3. **Custom Logic:** You can use getters and setters to run custom logic when accessing or modifying a property.

Static Methods

In JavaScript, **static methods** are methods that belong to the class itself, rather than instances of the class. This means that they are not called on instances but on the class itself. Static methods are often used for utility functions, helper methods, or operations that don't require an instance's data.

Syntax for Static Methods

```
class MyClass {  
  static myStaticMethod() {  
    console.log("This is a static method!");  
  }  
}
```

```
// Calling the static method directly on the class  
MyClass.myStaticMethod(); // Output: This is a static method!
```

Example of Static Methods

Here's a more practical example of using static methods in a class.

```
class Calculator {  
  // Static method to add two numbers
```

```
static add(a, b) {  
    return a + b;  
}  
  
// Static method to subtract two numbers  
static subtract(a, b) {  
    return a - b;  
}  
  
// Static method to multiply two numbers  
static multiply(a, b) {  
    return a * b;  
}  
  
// Static method to divide two numbers  
static divide(a, b) {  
    if (b === 0) {  
        console.log("Error: Division by zero");  
        return null;  
    }  
    return a / b;  
}  
  
// Calling static methods without creating an instance of the class  
console.log(Calculator.add(5, 3)); // Output: 8  
console.log(Calculator.subtract(5, 3)); // Output: 2  
console.log(Calculator.multiply(5, 3)); // Output: 15  
console.log(Calculator.divide(5, 3)); // Output: 1.6666666666666667
```

Key Characteristics of Static Methods:

- **No Instance Required:** Static methods are called on the class itself, not on instances of the class.
- **Access Class Properties:** They can access static properties or other static methods of the class.
- **Cannot Access Instance Properties:** They cannot access instance properties or methods, as they don't have access to `this` (which refers to the instance).

Why Use Static Methods?

1. **Utility Functions:** Static methods are useful for utility or helper functions that don't need to modify or rely on an instance's state.
2. **Class-Level Operations:** When you need to perform operations that pertain to the class as a whole, not to individual instances.
3. **Factory Methods:** Static methods can be used as factory methods to create instances of the class.

Example with Factory Method

Static methods can be used to create instances of the class based on certain conditions:

```
class User {
  constructor(name, role) {
    this.name = name;
    this.role = role;
  }

  // Instance method
  describe() {
    console.log(`${this.name} is a ${this.role}`);
  }

  // Static method as a factory method
  static createAdmin(name) {
    return new User(name, 'admin');
  }

  static createGuest(name) {
    return new User(name, 'guest');
  }
}

// Creating an instance using a static method
const admin = User.createAdmin('Alice');
const guest = User.createGuest('Bob');

// Calling the instance method 'describe' on the created instances
admin.describe(); // Output: Alice is a admin
guest.describe(); // Output: Bob is a guest
```

In this example, the static methods `createAdmin` and `createGuest` act as factory methods that create new `User` instances with predefined roles.

Object.create():

JavaScript `Object.create()` method is used to create a new object with the specified prototype object and properties. `Object.create()` method returns a new object with the specified prototype object and properties.

`Object.create(prototype, propertiesObject);`

prototype: The object that will be used as the prototype for the new object. This is what the new object will inherit from.

propertiesObject (optional): An object whose own enumerable properties will be added to the newly created object. This is like setting additional properties on the new object.

`Object.create()` in JavaScript is a method that creates a new object, using an existing object to provide the prototype for the newly created object. It allows you to set up inheritance and add properties or methods to the new object without modifying the original object.

Syntax:

`Object.create(prototype, propertiesObject);`

- **prototype:** The object that will be used as the prototype for the new object. This is what the new object will inherit from.
- **propertiesObject** (optional): An object whose own enumerable properties will be added to the newly created object. This is like setting additional properties on the new object.

Example:

```
// Create an object with a prototype
const person = {
  greet() {
    console.log("Hello!");
  }
};
```

```
// Create a new object with 'person' as its prototype
const john = Object.create(person);
```

```
// john now inherits the greet method from person
```

```
john.greet(); // Output: Hello!
```

Explanation:

- The `john` object has `person` as its prototype. This means `john` can access properties and methods defined in `person`, like the `greet` method.
- `john` doesn't directly have the `greet` method, but it can use it because of prototypal inheritance.

Coding Challenge #2

Your tasks:

1. Re-create Challenge #1, but this time using an ES6 class (call it 'CarCl')
2. Add a getter called 'speedUS' which returns the current speed in mi/h (divide by 1.6)
3. Add a setter called 'speedUS' which sets the current speed in mi/h (but converts it to km/h before storing the value, by multiplying the input by 1.6)
4. Create a new car and experiment with the 'accelerate' and 'brake' methods, and with the getter and setter.

Test data:

Data car 1: 'Ford' going at 120 km/h

```
//challenge2
class CarCl {
  constructor(make, speed) {
    this.make = make; // Car make (e.g., 'BMW')
    this.speed = speed; // Speed in km/h
  }

  accelerate() {
    this.speed += 10; // Increase speed by 10 km/h
    console.log(`${this.make} is going at ${this.speed} km/h`);
  }

  brake() {
    this.speed -= 5; // Decrease speed by 5 km/h
    console.log(`${this.make} is going at ${this.speed} km/h`);
  }
}
```

```

}

// Getter to convert speed to miles per hour (mph)
get speedUS() {
  return this.speed / 1.6; // Convert km/h to mph
}

// Setter to set speed in miles per hour and convert it to km/h
set speedUS(newSpeed) {
  this.speed = newSpeed * 1.6; // Convert mph to km/h
}
}

const bmw = new CarCl('BMW', 120); // Create BMW instance with speed 120 km/h
console.log(bmw); // Logs: CarCl { : 'BMW', speed: 120 }

const mercedes = new CarCl('Mercedes', 95); // Create Mercedes instance with speed 95 km/h
console.log(mercedes); // Logs: CarCl { : 'Mercedes', speed: 95 }

bmw.accelerate(); // BMW accelerates by 10 km/h
bmw.brake(); // BMW slows down by 5 km/h
mercedes.accelerate(); // Mercedes accelerates by 10 km/h
mercedes.brake(); // Mercedes slows down by 5 km/h

// Access current speed in miles per hour using getter
console.log(`The current speed of BMW in miles per hour is: ${bmw.speedUS}`);
console.log(
  `The current speed of Mercedes in miles per hour is: ${mercedes.speedUS}`
);

// Use setter to change the speed in miles per hour
bmw.speedUS = 80; // Set BMW's speed to 80 mph (will convert to km/h)
console.log(`BMW's new speed in km/h is: ${bmw.speed}`); // Logs new speed in km/h

```

```
► CarCl {make: 'BMW', speed: 120}
► CarCl {make: 'Mercedes', speed: 95}
BMW is going at 130 km/h
BMW is going at 125 km/h
Mercedes is going at 105 km/h
Mercedes is going at 100 km/h
The current speed of BMW in miles per hour is: 78.125
The current speed of Mercedes in miles per hour is: 62.5
BMW's new speed in km/h is: 128
```

Inheritance between classes

Before ES6, JavaScript used prototypes for inheritance. You can set the prototype of a child class to the parent class's prototype.

```
// Parent class
function Animal(name) {
  this.name = name;
}

Animal.prototype.speak = function () {
  console.log(this.name + ' makes a sound.');
```

```
// Child class inheriting from Animal
function Dog(name, breed) {
  Animal.call(this, name); // Call the parent class constructor
  this.breed = breed;
}

// Set Dog's prototype to be an instance of Animal
Dog.prototype = Object.create(Animal.prototype);
```

```
// Correct the constructor pointer
Dog.prototype.constructor = Dog;

Dog.prototype.speak = function () {
  console.log(this.name + ' barks. ');
};

// Creating an instance of Dog
const myDog = new Dog('Rex', 'Golden Retriever');
myDog.speak(); // Output: Rex barks.
```

In this approach:

- **Animal.call(this, name)**: Calls the parent class constructor within the child class.
- **Dog.prototype = Object.create(Animal.prototype)**: Sets up inheritance by linking the prototype of **Dog** to **Animal**.
- **Dog.prototype.constructor = Dog**: Corrects the constructor reference, which would otherwise point to **Animal**.

```
Dog.prototype = Object.create(Animal.prototype);
```

What happens is that we are creating a new object, **Object.create(Animal.prototype)**, which sets up the prototype chain so that objects created from **Dog** will inherit from **Animal**. This effectively means that **Dog.prototype** now inherits from **Animal.prototype**, but **Dog.prototype** is now an object created from **Animal.prototype**.

As a result:

- **Dog.prototype.constructor** is now **inherited from Animal.prototype** and points to **Animal**, not **Dog**.

This is problematic because when you create a new instance of **Dog** (like **new Dog()**), it will reference the **constructor** property of **Animal**, not **Dog**.

The Problem

If you directly check the constructor of an object created by **Dog**, like this:


```
const myDog = new Dog('Rex', 'Golden Retriever');  
console.log(myDog.constructor); // This will log 'Animal' instead of 'Dog'
```

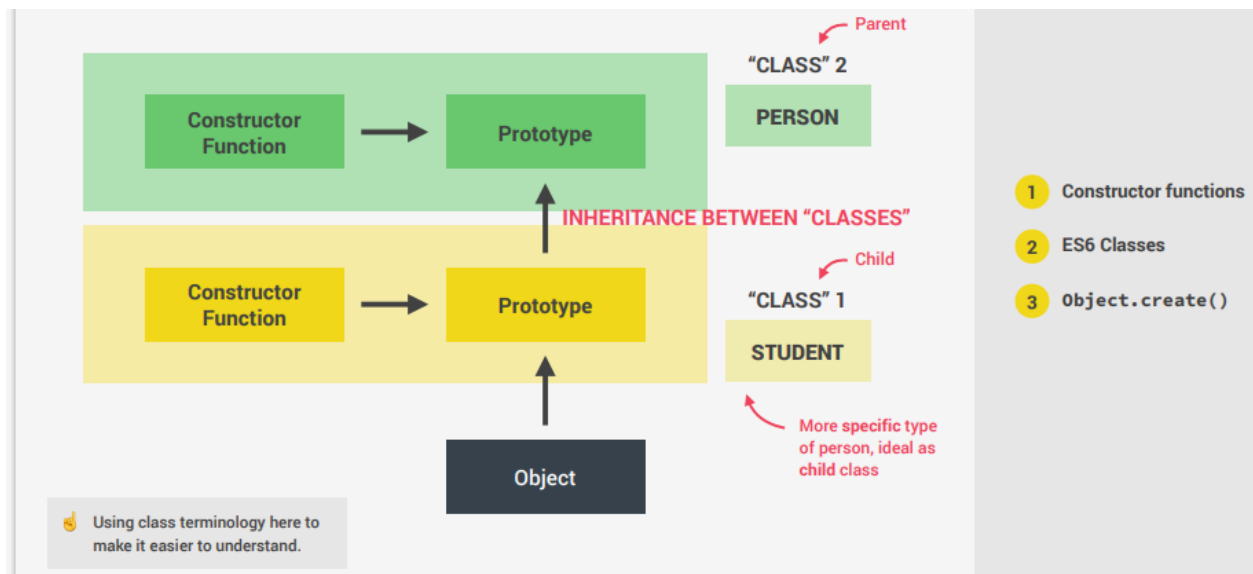
This will log **Animal** instead of **Dog** because the **constructor** property in **Dog.prototype** is pointing to **Animal**, not **Dog**.

The Solution: **`Dog.prototype.constructor = Dog;`**

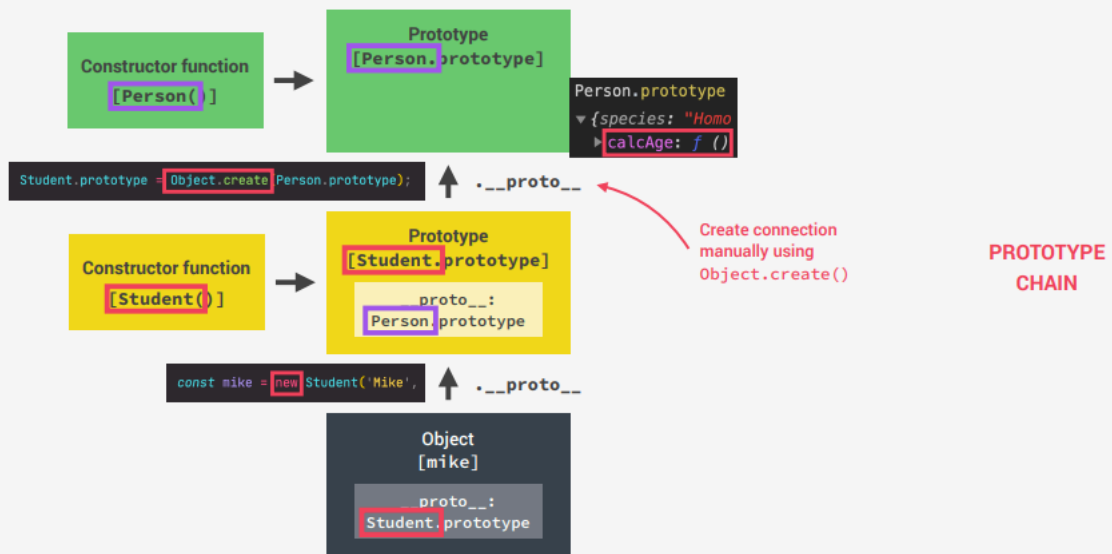
To fix this and ensure that the **constructor** property correctly points to **Dog**, we explicitly set it back:

```
Dog.prototype.constructor = Dog;
```

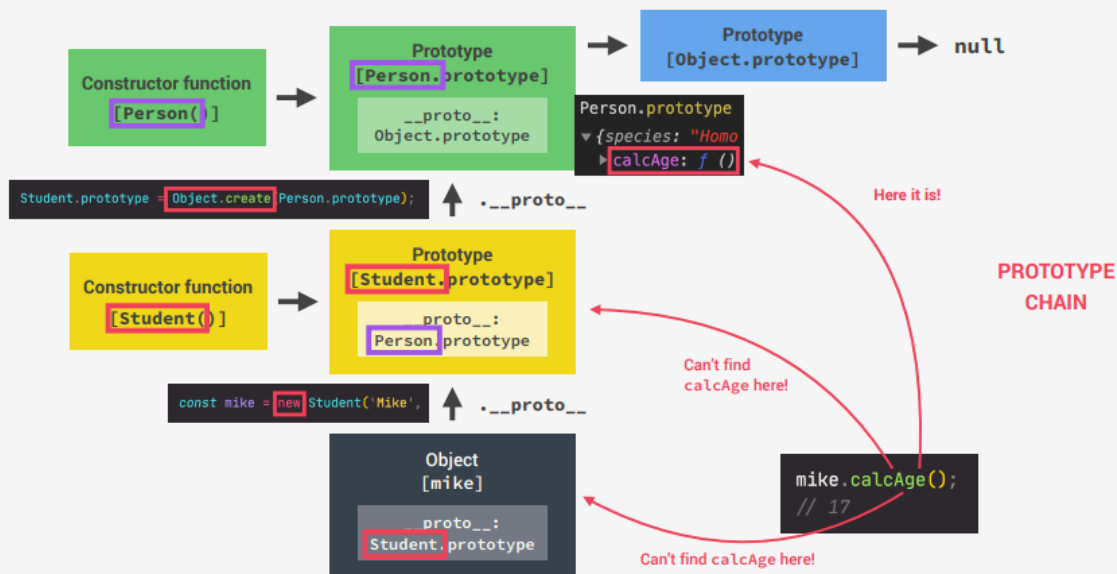
This line of code ensures that the **constructor** property on **Dog.prototype** points to the correct constructor function (**Dog**), not **Animal**.



INHERITANCE BETWEEN "CLASSES"



INHERITANCE BETWEEN "CLASSES"



Coding Challenge #3

Your tasks:

1. Use a constructor function to implement an Electric Car (called 'EV') as a child "class" of 'Car'. Besides a make and current speed, the 'EV' also has the current battery charge in % ('charge' property)
2. Implement a 'chargeBattery' method which takes an argument 'chargeTo' and sets the battery charge to 'chargeTo'
3. Implement an 'accelerate' method that will increase the car's speed by 20, and decrease the charge by 1%. Then log a message like this: 'Tesla going at 140 km/h, with a charge of 22%'
4. Create an electric car object and experiment with calling 'accelerate', 'brake' and 'chargeBattery' (charge to 90%). Notice what happens when you 'accelerate'! Hint: Review the definition of polymorphism 💡

Test data:

Data car 1: 'Tesla' going at 120 km/h, with a charge of 23%

```
// Car Constructor
const Car = function (make, speed) {
  this.make = make;
  this.speed = speed;
};
Car.prototype.accelerate = function () {
  this.speed += 10;
  console.log(`${this.make} is going at ${this.speed} km/h`);
};

// EV Constructor
const EV = function (make, speed, charge) {
  Car.call(this, make, speed); // Call the Car constructor to inherit properties
  this.charge = charge; // assign the 'charge' parameter to the EV object
};

//Linking Prototype properties
EV.prototype = Object.create(Car.prototype);
EV.prototype.constructor = EV; // constructor property correctly points to Dog,
//console.log(tesla.constructor);
```

```
// Create an instance of EV (Electric Vehicle)
EV.prototype.chargeBattery = function (chargeTo) {
  this.charge = chargeTo;
};
EV.prototype.accelerate = function (chargeTo) {
  this.speed += 20;
  this.charge--;
  console.log(
    `${this.make} going at ${this.speed} km/h, with a charge of ${this.charge}%`
  );
};

const tesla = new EV('Tesla', 120, 23);
console.log(tesla); // EV { make: 'Tesla', speed: 120, charge: 23 }
tesla.chargeBattery(90);
console.log(tesla); //EV {make: 'Tesla', speed: 120, charge: 90}
tesla.accelerate(); //Tesla going at 140 km/h, with a charge of 89%
```

When a method exists in both the parent and child classes, the method in the child class **overrides** the parent class's method is called Method Overriding. This is a key feature of JavaScript's prototype-based inheritance.

If the child class does not have the method, the parent class's method will be used.

inheritance using ES6 classes

example of inheritance in ES6, using an **Employee** class as the parent and a **Manager** class as the child:

Using **extends** and **super()** in ES6, we are achieving **classical inheritance** in JavaScript. This allows us to create a child class that inherits properties and methods from a parent class, while also enabling method overriding and additional functionality in the child class.

Key Components of Inheritance in ES6:

1. **extends** Keyword:

- Used to define a child class that inherits from a parent class.

- Links the prototype of the child class to the parent class, allowing the child class to inherit methods and properties.

2. **super()** Function:

- Used inside the child class constructor to call the parent class constructor.
- Must be called before accessing **this** in the child class constructor.
- Can also be used to call methods of the parent class.

Ex

```
// Parent Class: Employee
class Employee {
  constructor(name, id, salary) {
    this.name = name;
    this.id = id;
    this.salary = salary;
  }

  getDetails() {
    return `Name: ${this.name}, ID: ${this.id}, Salary: ${this.salary}`;
  }

  work() {
    console.log(`${this.name} is working.`);
  }
}

// Child Class: Manager
class Manager extends Employee {
  constructor(name, id, salary, department) {
    super(name, id, salary); // Call the Employee constructor
    this.department = department;
  }

  getDetails() {
    // Override the parent method
    return `${super.getDetails()}, Department: ${this.department}`;
  }
}
```

```

}

manage() {
  console.log(`${this.name} is managing the ${this.department} department.`);
}
}

// Create an Employee instance
const employee = new Employee('ram', 101, 50000);
console.log(employee.getDetails()); // Output: Name: ram, ID: 101, Salary: $50000
employee.work(); // Output: ram is working.

// Create a Manager instance
const manager = new Manager('Bob', 102, 80000, 'Sales');
console.log(manager.getDetails()); // Output: Name: Bob, ID: 102, Salary: $80000, Department: Sales
manager.work(); // Output: Bob is working.
manager.manage(); // Output: Bob is managing the Sales department.

```

Inheritance using Object.Create()

`Object.create()` keeps things simple and relies on JavaScript's prototype system directly. No constructors, no `new`, no `extends`—just objects linked to other objects.

Prototype Linking:

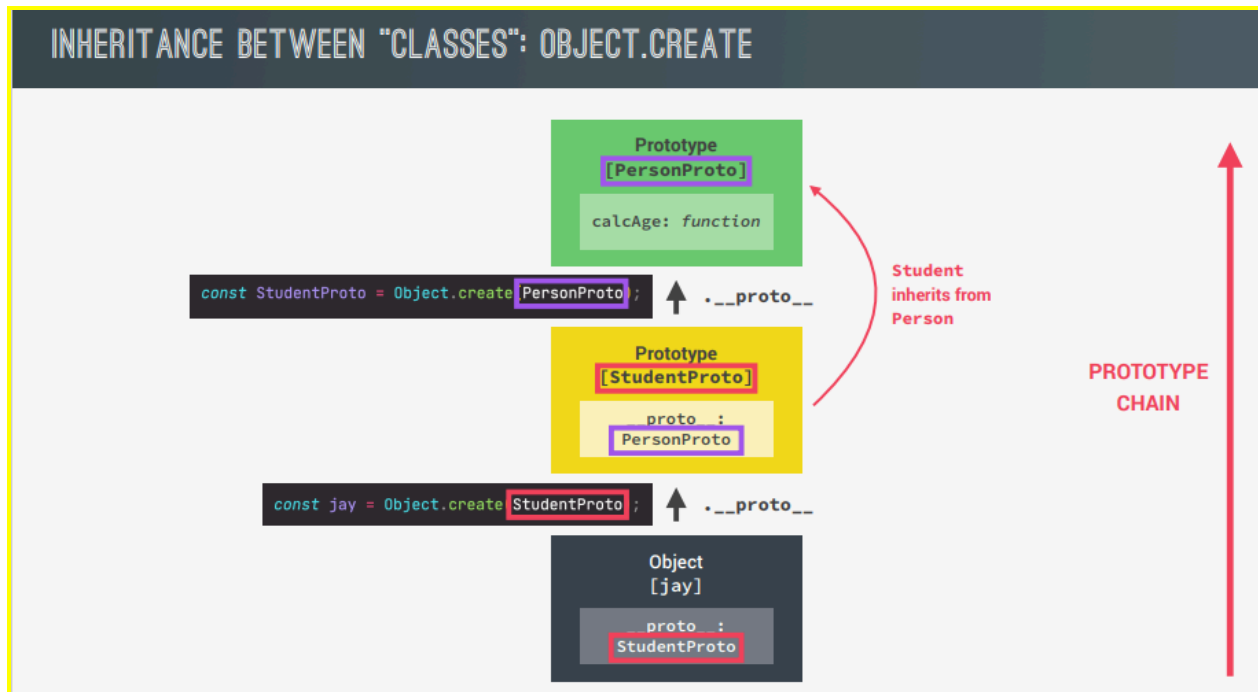
- `Object.create()` directly links the prototype of a new object to an existing object.
- This means the new object "inherits" all properties and methods from the prototype object.

No Constructors:

- Instead of constructors, you typically define an `init` method (or equivalent) for initializing the object's properties.
- This avoids the need for `super` calls or constructor chaining, as seen in ES6 classes.

No `new` Keyword:

- Objects are created with `Object.create()` instead of `new`. This eliminates the need for managing constructor functions explicitly.



Person → Student → Jay

```

// Base object: Person
const Person = {
  init(name, age) {
    this.name = name;
    this.age = age;
  },
  getDetails() {
    return `Name: ${this.name}, Age: ${this.age}`;
  },
};

// Derived object: Student
const Student = Object.create(Person);

Student.initStudent = function (name, age, grade) {
  
```

```

    this.init(name, age); // Call the Person's init method
    this.grade = grade;
};

Student.getDetails = function () {
    return `${Person.getDetails.call(this)}, Grade: ${this.grade}`;
};

// Further Derived object: Jay
const Jay = Object.create(Student);

Jay.initJay = function (name, age, grade, specialization) {
    this.initStudent(name, age, grade); // Call Student's init method
    this.specialization = specialization;
};

Jay.getDetails = function () {
    return `${Student.getDetails.call(this)}, Specialization: ${
        this.specialization
    }`;
};

// Usage
const person = Object.create(Person);
person.init('Alice', 50);
console.log(person.getDetails()); // Output: Name: Alice, Age: 50

const student = Object.create(Student);
student.initStudent('Bob', 20, 'A');
console.log(student.getDetails()); // Output: Name: Bob, Age: 20, Grade: A

const jay = Object.create(Jay);
jay.initJay('Jay', 25, 'A+', 'Computer Science');

```



```
console.log(jay.getDetails()); // Output: Name: Jay, Age: 25, Grade: A+, Specialization: Computer Science
```

How the Inheritance Chain Works

1. Base Class: **Person**

- Represents a general person with properties like **name** and **age**.
- Provides a **getDetails()** method to retrieve these properties.

2. Derived Class: **Student**

- Inherits from **Person** using **Object.create(Person)**.
- Adds a **grade** property and overrides the **getDetails()** method to include grade information.

3. Further Derived Class: **Jay**

- Inherits from **Student** using **Object.create(Student)**.
- Adds a **specialization** property and overrides the **getDetails()** method to include specialization information.

Key Points of the Hierarchy

1. **Prototype Chain:**
 - **Jay** is linked to **Student**, which is linked to **Person**.
 - **Jay.getDetails()** calls **Student.getDetails()**, which in turn calls **Person.getDetails()**.
2. **Initialization:**
 - Each level of the hierarchy has its own initializer (**init**, **initStudent**, **initJay**).
 - Each initializer calls the parent's initializer to avoid code duplication.
3. **Method Overriding:**
 - Each derived class (**Student**, **Jay**) overrides **getDetails()** to add specific details while reusing the parent's implementation using **.call(this)**.

Hierarchy Overview

1. **Person:**
 - Properties: **name**, **age**
 - Methods: **getDetails**
2. **Student** (inherits from **Person**):

- Additional Property: `grade`
 - Overridden Method: `getDetails`
3. **Jay** (inherits from `Student`):
- Additional Property: `specialization`
 - Overridden Method: `getDetails`
-

Private class fields and Methods

This approach provides true encapsulation, hiding the internal workings of a class from the outside world and only exposing the necessary public methods or properties to interact with the object.

Private Fields:

Private fields are variables that can only be accessed and modified within the class where they are defined. They cannot be accessed directly from outside the class.

Private Fields (#): Accessible only inside the class. Declared with the `#` prefix.

Public Fields: Accessible from outside the class. Declared without any prefix.

Private Methods:

Private methods are functions that can only be called within the class they are defined in. They are not accessible from outside the class.

- **Private Methods (#):** Used for internal logic within the class. They cannot be called or accessed outside.
- **Public Methods:** Used to interact with the object and manipulate its fields.

Private Fields/Methods:

- Must be declared with a `#` prefix (e.g., `#balance`, `#validateAmount`).
- Declared at the top of the class, outside the constructor or methods.
- Cannot be accessed or modified outside the class.
- Subclasses also cannot access private fields/methods.

Public Fields/Methods:

- No special syntax, defined normally (e.g., `accountType`, `displayAccountInfo()`).

- Can be accessed and modified directly from outside the class.

```
class BankAccount {
  // Private fields
  #accountHolderName; // Name of the account holder
  #balance; // Private balance

  // Public fields
  bankName = 'National Bank'; // Shared public field
  accountType; // Account type (Savings/Checking)

  // Constructor
  constructor(accountHolderName, initialDeposit, accountType) {
    this.#accountHolderName = accountHolderName; // Set private field
    this.#balance = initialDeposit > 0 ? initialDeposit : 0; // Set balance
    this.accountType = accountType; // Set public field
    this.accountNumber = `AC${Math.floor(Math.random() * 1000000)}`; // Unique public field
  }

  // Public method: Deposit money
  deposit(amount) {
    if (this.#validateAmount(amount)) {
      this.#balance += amount; // Update private balance
      console.log(`Successfully deposited $$${amount}.`);
    } else {
      console.log('Invalid deposit amount.');
```

```

    } else {
        console.log('Invalid withdrawal amount or insufficient balance.');
```

```

    }
}

// Public method: Display account details
displayAccountInfo() {
    console.log(`Bank Name: ${this.bankName}`);
    console.log(`Account Number: ${this.accountNumber}`);
    console.log(`Account Holder: ${this.#accountHolderName}`);
    console.log(`Account Type: ${this.accountType}`);
    console.log(`Current Balance: $$${this.#balance}`);
}

// Private method: Validate transaction amount
#validateAmount(amount) {
    return typeof amount === 'number' && amount > 0;
}
}

// Usage Example
const account = new BankAccount('Ram', 0, 'Savings');

// Access public fields
console.log(account.bankName); // 'National Bank'
console.log(account.accountType); // 'Savings'

// Call public methods
account.deposit(500); // Deposit money
account.withdraw(300); // Withdraw money
account.displayAccountInfo(); // Show account info

// Trying to access private fields/methods
```

```
// console.log(account.#balance); // SyntaxError: Private field '#balance' must be declared in an enclosing class
// account.#validateAmount(100); // SyntaxError: Private field '#validateAmount' must be declared in an enclosing class
```

Execution Flow for Example

1. **Initialization:**
 - Private fields (`#accountHolderName`, `#balance`) are declared and initialized in the constructor.
 - Public fields (`bankName`, `accountType`) are initialized directly.
2. **Deposit (`account.deposit(500)`):**
 - Call the public `deposit()` method.
 - Inside `deposit()`, the private method `#validateAmount()` checks the input.
 - If valid, `#balance` is updated.
3. **Withdraw (`account.withdraw(300)`):**
 - Call the public `withdraw()` method.
 - `#validateAmount()` checks the input. If valid and sufficient balance exists, `#balance` is decremented.
4. **Display Account Info:**
 - Calls `displayAccountInfo()`, which accesses both public and private fields to display details.
5. **Access Denied for Private Members:**
 - Attempting to access `#balance` or `#validateAmount()` outside the class results in a `SyntaxError`.

Key Benefits of Public and Private Members

1. **Encapsulation:** Keeps sensitive data (like `#balance`) safe from accidental or malicious tampering.
2. **Controlled Access:** Public methods (like `deposit()` and `withdraw()`) ensure proper validation before modifying private fields.
3. **Clear API:** Exposes only the necessary fields/methods, making the class easier to use and maintain.

Chaining Methods

To enable method chaining, you need to return the current instance (**this**) from each method that modifies the object's state. This allows you to call multiple methods on the same object in a single statement.

Here's the BankAccount class modified to support method chaining:

Example: Bank Account with Method Chaining

```
class BankAccount {
  // Private fields
  #accountHolderName;
  #balance;

  // Public fields
  bankName = 'National Bank';
  accountType;

  // Constructor
  constructor(accountHolderName, initialDeposit, accountType) {
    this.#accountHolderName = accountHolderName;
    this.#balance = initialDeposit > 0 ? initialDeposit : 0;
    this.accountType = accountType;
    this.accountNumber = `AC${Math.floor(Math.random() * 1000000)}`;
  }

  // Public method to deposit money (returns this for chaining)
  deposit(amount) {
    if (this.#validateAmount(amount)) {
      this.#balance += amount;
      console.log(`Successfully deposited $$${amount}.`);
    } else {
      console.log('Invalid deposit amount.');
```

```
    return this; // Enable chaining
  }

  // Public method to withdraw money (returns this for chaining)
```

```

withdraw(amount) {
  if (this.#validateAmount(amount) && this.#balance >= amount) {
    this.#balance -= amount;
    console.log(`Successfully withdrew $$${amount}.`);
  } else {
    console.log('Invalid withdrawal amount or insufficient balance.');
```

```

    return this; // Enable chaining
  }
}

// Public method to display account information (returns this for chaining)
displayAccountInfo() {
  console.log(`Bank Name: ${this.bankName}`);
  console.log(`Account Number: ${this.accountNumber}`);
  console.log(`Account Holder: ${this.#accountHolderName}`);
  console.log(`Account Type: ${this.accountType}`);
  console.log(`Current Balance: $$${this.#balance}`);
  return this; // Enable chaining
}

// Private method to validate transaction amounts
#validateAmount(amount) {
  return typeof amount === 'number' && amount > 0;
}
}

// Example usage with chaining
const account = new BankAccount('John Doe', 1000, 'Savings');
```

```

account
  .deposit(500) // Deposit money
  .withdraw(300) // Withdraw money
  .displayAccountInfo(); // Display account details
```

Key Changes to Enable Method Chaining

1. Return **this**:

- In each public method (`deposit`, `withdraw`, `displayAccountInfo`), add `return this;` as the last statement.

- This allows subsequent method calls to act on the same object.

2. Usage:

Instead of calling each method separately, you can chain them like:

`account.deposit(500).withdraw(300).displayAccountInfo();`

Execution Flow with Chaining

1. `account.deposit(500):`
 - Updates the private `#balance` field.
 - Returns the current object (`account`).
2. `account.withdraw(300):`
 - Deducts the amount from `#balance` if valid.
 - Returns the current object (`account`).
3. `account.displayAccountInfo():`
 - Displays account details.
 - Returns the current object (`account`).

ES6 Class Summary

The diagram illustrates the ES6 Class Summary with annotations for various class features and their usage in a `Student` class example. The annotations are as follows:

- Public field (similar to property, available on created object)**: Points to `university = "University of Tschon";`
- Private fields (not accessible outside of class)**: Points to `#studyHours = 0;`
- Static public field (available only on class)**: Points to `static numSubjects = 10;`
- Call to parent (super) class (necessary with extend). Needs to happen before accessing this**: Points to `super(fullName, birthYear);`
- Instance property (available on created object)**: Points to `this.startYear = startYear;`
- Redefining private field**: Points to `this.#course = course;`
- Public method**: Points to `introduce() { console.log('I study ${this.#course} at ${this.university}'); }`
- Referencing private field and method**: Points to `this.#makeCoffe();` and `this.#studyHours += h;`
- Private method (⚠ Might not yet work in your browser. "Fake" alternative: _ instead of #)**: Points to `#makeCoffe() { return 'Here is a coffe for you ☺'; }`
- Getter method**: Points to `get testScore() { return this._testScore; }`
- Setter method (use _ to set property with same name as method, and also add getter)**: Points to `set testScore(score) { this._testScore = score <= 20 ? score : 0; }`
- Static method (available only on class. Can not access instance properties nor methods, only static ones)**: Points to `static printCurriculum() { console.log('There are ${this.numSubjects} subjects'); }`
- Creating new object with new operator**: Points to `const student = new Student('Jonas', 2020, 2037, 'Medicine');`
- Parent class**: Points to `class Student extends Person`
- Inheritance between classes, automatically sets prototype**: Points to the `extends` keyword.
- Child class**: Points to the `Student` class.
- Constructor method, called by new operator. Mandatory in regular class, might be omitted in a child class**: Points to the `constructor` method.
- Classes are just "syntactic sugar" over constructor functions**: Points to the `class` keyword.
- Classes are not hoisted**: Points to the `class` keyword.
- Classes are first-class citizens**: Points to the `class` keyword.
- Class body is always executed in strict mode**: Points to the `class` keyword.

Challenge 4

Your tasks:

1. Re-create Challenge #3, but this time using ES6 classes: create an 'EVCI' child class of the 'CarCI' class
2. Make the 'charge' property private
3. Implement the ability to chain the 'accelerate' and 'chargeBattery' methods of this class, and also update the 'brake' method in the 'CarCI' class. Then experiment with chaining!

Test data: Data car 1: 'Rivian' going at 120 km/h, with a charge of 23%

```
//challenge 4
// Car Constructor

class CarCI {
  constructor(make, speed) {
    this.make = make;
    this.speed = speed;
  }
  accelerate() {
    this.speed += 10;
    console.log(`${this.make} is going at ${this.speed} km/h`);
    return this;
  }
}

// EV Constructor
class EVCI extends CarCI {
  #charge;
  constructor(make, speed, charge) {
    super(make, speed); // Call the Car constructor to inherit properties
    this.#charge = charge; // assign the 'charge' parameter to the EV object
  }
  chargeBattery(chargeTo) {
    this.#charge = chargeTo;
    return this; // Enable chaining
  }
}
```

```

    }

    accelerate(chargeTo) {
      this.speed += 20;
      this.#charge--;
      console.log(
        `${this.make} going at ${this.speed} km/h, with a charge of ${
          this.#charge
        }%`
      );
      return this;
    }
  }

const car1 = new EVCI('Rivian', 120, 23);
console.log(car1); //{make: 'Rivian', speed: 120, #charge: 23}
// tesla.chargeBattery(90);
// console.log(tesla); //EV {make: 'Tesla', speed: 120, charge: 90}
// tesla.accelerate(); //Tesla going at 140 km/h, with a charge of 89%

//chaining
car1.accelerate().chargeBattery(200).accelerate();
//Rivian going at 140 km/h, with a charge of 22%
// Rivian going at 160 km/h, with a charge of 199%

```

In JavaScript, there are two common ways to make something "**private**" — meaning it's not supposed to be accessed directly from outside the class:

1. The # Symbol (True Privacy)

- This is a **new feature** in JavaScript that **enforces privacy**. When you use **#** before a property or method inside a class, it means that only the class itself can access or change that property.
- If you try to access it outside the class, you'll get an error.

Example:

```
class MyClass {
  #privateValue = 100; // This is a private property.

  getPrivateValue() {
    return this.#privateValue; // Only inside the class can we access it.
  }
}

const instance = new MyClass();
console.log(instance.getPrivateValue()); // 100 (works fine)
console.log(instance.#privateValue); // Error: Private field '#privateValue' must be declared in an
enclosing class
```

- The `#privateValue` cannot be accessed outside the class — it's really private!

2. The `_` Convention (Not Real Privacy)

- This is a **convention** where we use an underscore (`_`) before a property name to **suggest** that it should be private. But in reality, this is just a **warning** for other developers. JavaScript doesn't stop you from accessing the property, so it's not truly private.
- It's like saying, "Hey, don't touch this, it's private," but JavaScript won't stop you if you do.

Example:

```
class MyClass {
  constructor() {
    this._privateValue = 100; // This is just a suggestion, not true privacy.
  }

  getPrivateValue() {
    return this._privateValue; // We still can access it within the class.
  }
}

const instance = new MyClass();
console.log(instance.getPrivateValue()); // 100 (works fine)
console.log(instance._privateValue); // 100 (you can still access it directly)
```

- In this case, using `_privateValue` is just a **suggestion** to developers that this property shouldn't be accessed directly, but JavaScript itself won't block you from doing so.

Summary:

- **# (Hashtag): True privacy** — the property or method can only be accessed inside the class, not from outside. This is the best way to enforce privacy in modern JavaScript.
- **_ (Underscore): Just a convention** — it's a hint to other developers to avoid using the property directly, but JavaScript doesn't actually make it private.