## **My "Eureka!" Moment: Finally Understanding JavaScript Prototype Inheritance and Classes**

For years, JavaScript's prototype inheritance, classes, and super() felt like an unsolvable puzzle to me. No matter how many tutorials I read, I couldn’t fully grasp what was happening behind the scenes. The concept of prototypical inheritance seemed distant and abstract, and the class syntax in ES6 only confused me more. I saw classes in JavaScript as something similar to classes in Java or Python, and every time super() was mentioned, I felt lost.

But finally, after breaking it down step by step, **it clicked!** And now, I want to share my insights with fellow developers who might be struggling with the same concepts. Hopefully, my journey to understanding prototype inheritance will help others have their own "aha!" moment.

### **The Common Misconception About JavaScript Classes**

Many JavaScript developers (myself included) tend to approach class in JavaScript as if it works like traditional object-oriented programming (OOP) languages such as Java or C#. But here’s the truth:

🔹 **JavaScript classes are just syntactic sugar over prototypical inheritance.** 🔹 **Behind the scenes, everything in JavaScript is still prototype-based.** 🔹 **When you create a class in JavaScript, you’re essentially creating a constructor function that sets up prototype-based inheritance.**

Once I understood this, I realized that I had been looking at JavaScript the wrong way. JavaScript doesn’t have "real" classes like Java; it just has an elegant way to structure prototype-based inheritance using a familiar OOP-like syntax.

### **Understanding Prototype Inheritance**

Before ES6 introduced class, JavaScript relied entirely on prototype chains for inheritance. Here’s how it works:

const parent = {

greet() {

return "Hello from Parent!";

}

};

const child = Object.create(parent); // Child object inherits from parent

console.log(child.greet()); // "Hello from Parent!" ✅ Inherited

This is **prototypical inheritance in action**:

* child doesn’t have a greet() method, but JavaScript looks up the prototype chain and finds it in parent.
* Object.create(parent) links child to parent, making parent the prototype of child.

With ES6 classes, we now have a more structured way to write this:

class Parent {

greet() {

return "Hello from Parent!";

}

}

class Child extends Parent {}

const instance = new Child();

console.log(instance.greet()); // "Hello from Parent!" ✅ Inherited

But under the hood, JavaScript is still using prototype chains!

### **Prototype Shadowing: When a Child Class Overrides a Method**

If a child class defines a method with the same name as the parent, it **shadows** the parent method.

class Parent {

greet() {

return "Hello from Parent!";

}

}

class Child extends Parent {

greet() {

return "Hello from Child!"; // ✅ Shadows Parent method

}

}

const instance = new Child();

console.log(instance.greet()); // "Hello from Child!" ✅ Shadowed

Here, JavaScript first looks for greet() in Child. Since it finds one, it **never checks the parent class**. The parent's greet() method is completely hidden unless we explicitly call it using super().

### **Finally Understanding super()**

This was the part that used to confuse me the most. Why do we need super()? What does it actually do?

🔹 In a **child class**, this **is not initialized** until super() is called.  
 🔹 **Calling super() invokes the parent’s constructor and sets up this.** 🔹 **If you try to use this before calling super(), JavaScript throws an error.**

#### **❌ Example That Fails Without super()**

class Parent {

constructor(name) {

this.name = name;

}

}

class Child extends Parent {

constructor(age) {

this.age = age; // ❌ ERROR: Must call super() before using 'this'

}

}

const child = new Child(3); // ❌ ReferenceError

🔴 **Error:** "ReferenceError: Must call super constructor in derived class before accessing 'this'"

#### **✅ Corrected Version With super()**

class Parent {

constructor(name) {

this.name = name;

}

}

class Child extends Parent {

constructor(name, age) {

super(name); // ✅ Calls Parent constructor first

this.age = age; // ✅ Now `this` is safe to use

}

}

const child = new Child("Noah", 3);

console.log(child.name); // "Noah" ✅ Inherited from Parent

console.log(child.age); // 3 ✅ Defined in Child

**Takeaways:**

1. super() **resets this** so that it refers to the child instance, not the parent.
2. **If your class extends another class and has a constructor, you MUST call super() before using this.**
3. If your class **does not extend** another class, super() is not needed.

### **Final Thoughts: My Journey to Understanding JavaScript Inheritance**

It took me **years** to finally have this "Eureka!" moment. The key realization was that:

* **JavaScript classes are just syntactic sugar over prototype inheritance.**
* **Everything in JavaScript inheritance happens through prototype chains.**
* **Understanding super() is key to working with class-based inheritance.**

I feel like many developers struggle with this topic because they don’t fully grasp the connection between JavaScript's prototype-based inheritance and the class syntax. I hope sharing my journey helps other developers reach their own "aha!" moment faster than I did!

🚀 **Have you struggled with prototype inheritance in JavaScript? What was your breakthrough moment? Let's discuss in the comments!**