

Lecture 21: Prototype and Class in javascript

Part 1: The "Why" - The Fundamental Problem of Repetition

- **The Atom:** The JavaScript Object. What is it at its core? A simple collection of key-value pairs.
- **The Problem:** We'll create a few simple, similar objects and immediately see the inefficiency. We're duplicating functions (code) in memory and making updates a nightmare. This establishes the *need* for a better way.

Part 2: The First Principle Solution - The Prototype Chain

- **The Core Truth:** In JavaScript, objects don't have copies of methods; they have a *link* to another object. This is the single most important concept.
- **The `[[Prototype]]` Link:** We'll explore this hidden, internal link that every object has. This is the physical mechanism for inheritance.
- **The Chain Reaction:** What happens when you try to access a property (`obj.sayHi`)? We'll trace the lookup process up the "prototype chain" step-by-step. This is the *behavior* that results from the `[[Prototype]]` link.

Part 3: Creating the Links - The "How" (The Old Way)

- **The Blueprint Maker:** The Constructor Function. We'll see how regular functions can be used with the `new` keyword to manufacture objects.
- **Demystifying `new` :** We'll break down the four things the `new` keyword does automatically. This is where the magic happens, and we're going to expose the trick.
- **The `prototype` Property:** We'll uncover the crucial difference between an object's hidden `[[Prototype]]` link and a function's public `.prototype` property. This is a common point of confusion that we will make crystal clear.

Part 4: The Modern Abstraction - JavaScript `class`

- **The "Syntactic Sugar":** We'll introduce the `class` keyword.

- **Lifting the Veil:** We will directly compare a `class` implementation to the constructor function implementation from Part 3. We'll prove, with code, that the `class` syntax is just a cleaner, more convenient way to do the *exact same thing* we learned in Part 3. It still creates constructor functions and manipulates the prototype chain underneath.
- **Inheritance with `extends`:** We'll briefly touch on how the `extends` keyword is just a convenient shortcut for manually setting up the prototype chain between two constructor functions.

By the end of this, you won't just know *how* to use prototypes and classes; you will understand *what they are* at a fundamental level. You'll be able to reason about object behavior instead of guessing.

Ready to begin with Part 1?

Part 1: The "Why" - The Fundamental Problem of Repetition

Let's start with the most basic building block in JavaScript: an object. At its core, an object is just a bag of properties (key-value pairs).

Imagine we're creating a simple game with users.

```
let user1 = { name: "Alice", score: 100, sayHi: function() { console.log("Hi, I'm " + this.name); } }; let user2 = { name: "Bob", score: 50, sayHi: function() { console.log("Hi, I'm " + this.name); } }; user1.sayHi(); // Outputs: Hi, I'm Alice user2.sayHi(); // Outputs: Hi, I'm Bob
```

Look at that code. Now, let's analyze it from first principles.

What's the problem?

1. **Inefficiency (Memory):** The `sayHi` function is identical for both users. Right now, JavaScript is creating and storing two separate, identical copies of that function in memory. If we have 1,000,000 users, we will have 1,000,000 copies of the exact same function. That's a huge waste.
2. **Inefficiency (Maintenance):** What if we want to add a new method, `increaseScore()`, to all users? We'd have to go back and manually add it to `user1`, `user2`, and every other user object we've created. What if we want to fix a bug in `sayHi`? We'd have to fix it in every single object. This is a nightmare.

The Fundamental Question:

How can we let multiple objects share a method without owning a copy of it?

The answer to this question is the reason prototypes exist. We need a central place to store the shared stuff, and a way for our user objects to find it.

This leads us directly to the core mechanism. Ready for Part 2?

Part 2: The First Principle Solution - The Prototype Chain

The creators of JavaScript solved the problem from Part 1 with a simple, powerful idea: **objects can be linked to other objects**.

Let's call this central "shared stuff" object `userFunctions`.

```
// A central place for all shared methods const userFunctions = { sayHi: function() { console.log("Hi, I'm " + this.name); }, increaseScore: function() { this.score++; } };
```

Now, how do we connect our individual user objects to `userFunctions` so they can use `sayHi` ?

This is where the secret link comes in. Every object in JavaScript has a hidden internal property, officially called `[[Prototype]]`. You can think of it as a secret "parent" or a "fallback" object.

When you try to access a property on an object, here's what JavaScript does:

1. Does the object itself have this property? (`user1.name` ? Yes, it's "Alice").
2. If not, does its `[[Prototype]]` object have this property?
3. If not, does *that* object's `[[Prototype]]` object have the property?
4. This continues all the way up until it finds a `[[Prototype]]` that is `null`. This sequence of linked objects is called the **prototype chain**.

Let's create this link manually using `Object.create()`, a method that creates a new object and sets its `[[Prototype]]` to an object you provide.

```
// Shared methods const userFunctions = { sayHi: function() { console.log("Hi, I'm " + this.name); } }; // Create a new user, but tell it to use userFunctions as its fallback/prototype let user1 = Object.create(userFunctions); user1.name = "Alice"; user1.score = 100; let user2 = Object.create(userFunctions); user2.name = "Bob"; user2.score = 50; // LET'S TEST IT user1.sayHi(); // "Hi, I'm Alice"
```

What just happened when we called `user1.sayHi()` ?

1. JavaScript looked for `sayHi` on the `user1` object. It couldn't find it. `user1` only has `name` and `score`.
2. JavaScript didn't give up. It followed the hidden `[[Prototype]]` link from `user1` to the `userFunctions` object.
3. It looked for `sayHi` on the `userFunctions` object. It found it!
4. It executed the function. Crucially, the value of `this` inside the function was still `user1`, the object that *initiated the call*.

We have solved the problem! We have one copy of the `sayHi` function, and both `user1` and `user2` can use it. This is prototypal inheritance in its purest form.

Let that sink in. This is the bedrock. Everything else is just a more convenient way of creating this link.

Now, `Object.create()` is a bit clunky. For years, JavaScript developers used a more common pattern to automate this process. Ready for Part 3?

Part 3: Creating the Links - The "How" (The Old Way)

Creating objects with `Object.create()` and then adding properties one-by-one works, but it's tedious. The community developed a standard pattern using regular functions.

The Constructor Function

Any function in JavaScript can become a "constructor function" if you call it with the `new` keyword. By convention, we capitalize their names to signal that they are meant to be used this way.

```
function User(name, score) { this.name = name; this.score = score; } // No
w, we need to put the shared methods somewhere. // Where do we put them?
// Every function in JavaScript automatically gets a special public proper
ty called 'prototype'. // This is NOT the hidden [[Prototype]]. This is a
plain object. User.prototype.sayHi = function() { console.log("Hi, I'm " +
this.name); }; User.prototype.increaseScore = function() { this.score++;
}; // Now, we use the 'new' keyword const user1 = new User("Alice", 100);
const user2 = new User("Bob", 50); user1.sayHi(); // "Hi, I'm Alice" user
2.increaseScore(); console.log(user2.score); // 51
```

This looks very different, but it achieves the *exact same result* as Part 2. The `new` keyword is the key.

Demystifying the `new` keyword

When you call `new User("Alice", 100)`, four things happen automatically behind the scenes:

1. **Create Empty Object:** An empty object is created, like `{}`.
2. **Link Prototype:** This new object's hidden `[[Prototype]]` is linked to the constructor function's `prototype` object (`User.prototype`).
 - This is the most critical step! This is how `user1` gets connected to the shared methods. `user1.[[Prototype]]` now points to `User.prototype`.
3. **Execute Constructor:** The `User` function is called, and `this` is set to point to the newly created object. So `this.name = name;` is actually `newObject.name = "Alice";`.
4. **Return `this`:** The function implicitly returns the new object (`this`).

So, `new` is just an automated recipe for creating an object and setting up its prototype link for us.

Recap of the two "prototypes":

- `user1.[[Prototype]]` (or `user1.__proto__`): The *actual hidden link* on an object that points to its prototype (its parent/fallback).
- `User.prototype`: A regular object that sits on a *constructor function*. It's the object that will be assigned as the `[[Prototype]]` for all instances created with `new User()`.

Understanding this distinction is the key to mastering prototypes.

This pattern worked for over a decade, but it's a bit weird, right? The methods are defined outside the function body, and the `prototype` property is a bit confusing. Programmers coming from languages like Java or Python wanted a cleaner syntax.

This leads us to the final evolution. Ready for Part 4?

Part 4: The Modern Abstraction - JavaScript `class`

The `class` keyword was introduced in ES2015 to make the process from Part 3 look nicer and more familiar.

Let me be clear: `class` does not introduce a new object model to JavaScript. It is "syntactic sugar" over the existing prototypal inheritance system. It does the *exact same thing* as the constructor function pattern, just with a cleaner syntax.

Let's rewrite our `User` from Part 3 using the `class` syntax.

```
class User { constructor(name, score) { this.name = name; this.score = score; } // Methods defined here are automatically put on the prototype! sayHi() { console.log("Hi, I'm " + this.name); } increaseScore() { this.score++; } } const user1 = new User("Alice", 100); const user2 = new User("Bob", 50); user1.sayHi(); // "Hi, I'm Alice" console.log(user2.score); // 50 user2.increaseScore(); console.log(user2.score); // 51
```

Let's break this down and prove it's the same thing:

- The `constructor` method is the same as our old `function User(...)`. It's what gets called when you use `new`.
- Any other methods you write inside the class (`sayHi`, `increaseScore`) are automatically placed on the `User.prototype` object for you. No more typing `User.prototype.whatever = ...`. It's a convenient shortcut.

The Proof:

Let's run some checks to see what's happening under the hood.

```
// A class is really just a special kind of function console.log(typeof User); // "function" // The methods are on the prototype, not the instance console.log(user1.hasOwnProperty('sayHi')); // false console.log(User.prototype.hasOwnProperty('sayHi')); // true // The instance's [[Prototype]] is linked to the class's .prototype console.log(Object.getPrototypeOf(user1) === User.prototype); // true
```

The evidence is clear. The `class` syntax is a clean, modern wrapper around the constructor function and prototype mechanism we built from first principles. It even has nice features like `extends` for making one class's prototype link to another, which used to be a very messy manual process.

Final Summary from First Principles

1. **Problem:** Creating many similar objects is inefficient because you duplicate shared functions in memory.
2. **Fundamental Solution:** Create a link (`[[Prototype]]`) from an instance object to a shared "prototype" object.

3. **Mechanism:** When you access a property, JavaScript checks the instance first, then follows the `[[Prototype]]` link up the chain until it finds the property or the chain ends.
4. **Old Implementation:** Use a "Constructor Function" with the `new` keyword, which automates the creation of a new object and links its `[[Prototype]]` to the function's `.prototype` property.
5. **Modern Abstraction:** Use the `class` keyword, which provides a much cleaner syntax for doing the exact same thing as the constructor function pattern.

THIS Keyword

The `this` keyword is a special identifier in JavaScript. It doesn't have a fixed value. Instead, its value is determined by **how the function is called**. It's a reference to the **context** of the call.

In a `constructor`, the `this` keyword has a very specific and crucial job.

The Analogy: The Blank Nametag

Imagine you're at a conference registration desk.

1. The `constructor` is the **registration process**.
2. The parameters (`name` , `score`) are the **details you provide** on the registration form.
3. The `new` keyword hands the constructor a **brand new, blank nametag**. This blank nametag is `this`.

The job of the constructor is to take the details from the form and write them onto that blank nametag.

So when the code runs:

```
constructor(name, score) { // this.name = name; // this.score = score; }
```

You can read it like this:

- `this.name = name;` means: "Take the `name` ('Alice') from the form and write it in the 'Name' field of **this specific nametag**."

- `this.score = score;` means: "Take the `score` (100) from the form and write it in the 'Score' field of **this specific nametag**."

At the end of the process, the constructor hands back the fully filled-out nametag, which is now a complete object.

The Technical Explanation (What `new` Does)

Let's revisit the four things the `new` keyword does when you call `new User("Alice", 100)`:

1. **Creates a brand new, empty object.** Let's call it `newInstance = {}`.
2. **Links the prototype.** It sets `newInstance`'s prototype to `User.prototype`.
3. **Calls the constructor.** It calls the `constructor` function, and—this is the most important part—it sets the `this` keyword to be `newInstance`. So, inside the constructor, `this` is a direct reference to the empty object created in step 1.
4. **Returns `this`.** The `new` expression returns the newly constructed object (`newInstance`), which is now populated with properties.

So, `this` is the mechanism that allows the constructor to **populate the new object**.

What Would Happen Without `this` ?

If you tried to write the constructor without `this`, it would fail completely:

```
class User { constructor(name, score) { // This is WRONG and does not work
// 'name' here just refers to the local parameter, not a property. name =
name; score = score; // We never attached anything to the new object being
created. } } const user1 = new User("Alice", 100); console.log(user1.name);
// undefined console.log(user1.score); // undefined
```

The `user1` object would be created, but it would be empty because we never used `this` to attach the `name` and `score` properties to it. We just had local variables that disappeared when the constructor finished running.