Open in app ↗

Medium Search

Get unlimited access to the best of Medium for less than $1/week. Become a member

Understanding Prototypes in JavaScript: The Backbone of Inheritance

4 min read · Dec 23, 2024

Mostafizur Rahman Follow

Listen Share More

Introduction

JavaScript is an incredibly versatile language, and one of the key concepts that sets it apart is its prototype-based inheritance model. If you're a JavaScript developer or someone transitioning from a class-based object-oriented language, understanding prototypes is essential. In this article, we'll break down the concept of prototypes, explore how they work under the hood, and dive into some examples to make this topic crystal clear. By the end of this post, you'll understand why prototypes are powerful and how to use them effectively in your code.

What Are Prototypes in JavaScript?

In JavaScript, every object has a hidden property called `Prototype` that either points to another object or is `null`. This prototype chain allows objects to inherit properties and methods from other objects. Essentially, prototypes are the foundation of inheritance in JavaScript.

An Example: The "Superhero Blueprint"

Let's say we are building a game with superheroes, and we want to create a blueprint for all superhero objects.

```
// Create a prototype object
const superheroPrototype = {
```

```
    fly() {
      console.log(`${this.name} is flying!`);
    },
    fight() {
      console.log(`${this.name} is fighting with ${this.power}!`);
    },
  };

  // Create a superhero object
  const ironMan = Object.create(superheroPrototype);
  ironMan.name = "Iron Man";
  ironMan.power = "Repulsor Beams";

  // Create another superhero object
  const thor = Object.create(superheroPrototype);
  thor.name = "Thor";
  thor.power = "Mjolnir";

  ironMan.fly(); // Output: Iron Man is flying!
  ironMan.fight(); // Output: Iron Man is fighting with Repulsor Beams!

  thor.fly(); // Output: Thor is flying!
  thor.fight(); // Output: Thor is fighting with Mjolnir!
```

In the example above:

1. The `superheroPrototype` serves as a common blueprint.
2. `Object.create(superheroPrototype)` creates new objects linked to the prototype.
3. Each superhero object has its own properties (`name` and `power`) while inheriting methods (`fly` and `fight`) from the prototype.

How Does the Prototype Chain Work?

When you try to access a property or method on an object, JavaScript will:

1. Look for the property directly on the object.
2. If it doesn't find it, it will check the object's prototype.
3. If the property still isn't found, it will traverse up the prototype chain until it reaches `null`.

Example: Exploring the Prototype Chain

```
console.log(ironMan.__proto__); // Logs superheroPrototype
console.log(superheroPrototype.__proto__); // Logs Object.prototype
console.log(Object.prototype.__proto__); // Logs null
```

This process is known as **prototype chaining**, and it's what makes inheritance in JavaScript work.

Prototypes in Custom Constructors

You can also use prototypes to define methods for custom constructors. Let's extend our superhero example with a constructor function:

```
function Superhero(name, power) {
  this.name = name;
  this.power = power;
}

// Adding methods to the prototype
Superhero.prototype.fly = function () {
  console.log(`${this.name} is flying!`);
};

Superhero.prototype.fight = function () {
  console.log(`${this.name} is fighting with ${this.power}!`);
};

// Creating new superhero instances
const hulk = new Superhero("Hulk", "Super Strength");
const captainAmerica = new Superhero("Captain America", "Vibranium Shield");

hulk.fly(); // Output: Hulk is flying!
captainAmerica.fight(); // Output: Captain America is fighting with Vibranium S
```

In this example:

- The `Superhero` function acts as a constructor for new superheroes.
- Methods (`fly` and `fight`) are added to `Superhero.prototype`, ensuring they are shared across all instances.

Breaking Misconceptions: Prototypes vs Classes

Many developers new to JavaScript often confuse prototypes and classes. Let's clarify:

- **Prototypes:** JavaScript uses prototypes as its core inheritance model. Even with ES6 classes, prototypes are working under the hood.
- **Classes:** They are syntactic sugar for prototypes. The following class example is equivalent to the custom constructor example above:

```
class Superhero {
  constructor(name, power) {
    this.name = name;
    this.power = power;
  }

  fly() {
    console.log(`${this.name} is flying!`);
  }

  fight() {
    console.log(`${this.name} is fighting with ${this.power}!`);
  }
}

const spiderman = new Superhero("Spider-Man", "Web Slinging");
spiderman.fly(); // Output: Spider-Man is flying!
```

When Should You Use Prototypes Directly?

Prototypes are powerful, but when should you use them directly instead of relying on classes or libraries?

1. **Memory Efficiency:** Methods defined on prototypes are shared among all instances, reducing memory usage compared to defining methods in each object.
2. **Custom Object Creation:** Use prototypes when you need more control over object creation (e.g., `Object.create` for custom inheritance chains).
3. **Browser Compatibility:** For older browsers without ES6 class support, prototypes provide a robust way to implement inheritance.

Best Practices for Using Prototypes

1. **Use Prototypes for Shared Methods:** Define methods on prototypes to keep your code DRY (Don't Repeat Yourself).
2. **Avoid Overriding Built-In Prototypes:** Modifying native prototypes like `Array.prototype` can lead to unexpected behavior and compatibility issues.
3. **Use `Object.create` for Prototypal Inheritance:** It's a clean and efficient way to create objects with specific prototypes.

Example: Extending Built-In Objects (With Caution)

```
Array.prototype.first = function () {
  return this[0];
};

const numbers = [10, 20, 30];
console.log(numbers.first()); // Output: 10
```

While this works, be cautious — other libraries or scripts may rely on the default behavior of `Array.prototype`.

Conclusion

Understanding prototypes is key to mastering JavaScript. Whether you're building complex applications or simply optimizing your code, leveraging prototypes effectively can make your code more efficient and maintainable. By combining prototype chaining, custom constructors, and best practices, you'll unlock the full potential of JavaScript's inheritance model.

JavaScript | Javascript Tips | Javascript Development | Javascript Interview | Frontend Development

Follow