JavaScript Objects and Prototypes In-depth

> Comprehensive notes on the JavaScript Objects and Prototypes In-depth series by Java Brains. This resource dives deep into JavaScript's core concepts, including objects, prototypes, inheritance, and the prototype chain, with clear explanations and practical examples. Perfect for mastering the foundations of JavaScript object-oriented programming!

📺 Watch on YouTube

Unit 01 - Creating Objects

01 - Introduction

This course explores JavaScript’s object-oriented programming, covering object creation, constructors, execution contexts, prototypes, and practical coding examples.

Highlights

- 🏗️ Introduction to object-oriented programming in JavaScript.
- 🔧 Various methods for creating objects in JavaScript.
- 🏛️ Understanding constructors and their unique JavaScript implementation.
- 🔄 Exploring function execution and its reference context.
- 📜 Demystifying prototypes and their significance in JavaScript.
- 📚 Learning to build object blueprints without traditional classes.
- 💻 Hands-on coding examples to reinforce concepts learned.

Note: The ES6 version has a class keyword that gives class like syntax but we still don’t get all the typical class features.

02 - Objects Basics

JavaScript objects are collections of values, allowing for flexible structures and properties. They can be created in various ways, including inline.

Highlights

- 📦 Objects are collections of multiple values.
- 🎨 Objects can contain various data types, including numbers, strings, arrays, and functions.
- ✏️ Objects are created using curly braces `{}`; an empty object is made with `{}`.
- 🔑 Properties can be added dynamically, allowing for a flexible structure.
- 📊 Access properties using dot notation (`obj.prop`) or bracket notation (`obj['prop']`).
- 🏠 Nested objects are possible, allowing for complex data structures.
- 📚 Further learning is encouraged for a deeper understanding of JavaScript objects.

03 - Creating Objects

Over here we learnt how to create employee objects with properties such as `firstName`, `lastName`, `gender`, and `designation`.

```
const emp1 = {
  firstName: "John",
```

```
  lastName: "Doe",
  gender: "Male",
  designation: "Software Engineer",
};
const emp2 = {
  firstName: "Jane",
  lastName: "Smith",
  gender: "Female",
  designation: "Project Manager",
};
```

However, manually creating multiple employee objects could quickly become repetitive and inefficient.

To address this, let's introduce a reusable function called `createEmployeeObject`, which accepts parameters for each property and dynamically generates employee objects.

```
function createEmployeeObject(firstName, lastName, gender, designation) {
  var newObj = {};
  newObj.firstName = firstName;
  newObj.lastName = lastName;
  newObj.gender = gender;
  newObj.designation = designation;

  return newObj;
}

// Using the function to create employee objects
const emp1 = createEmployeeObject("John", "Doe", "Male", "Software Engineer");
const emp2 = createEmployeeObject("Jane", "Smith", "Female", "Project Manager");
const emp3 = createEmployeeObject("Emily", "Taylor", "Female", "QA Tester");
```

This approach avoids the redundancy of manually defining each object, as shown in the earlier examples of `emp1` and `emp2`, and allows to efficiently create additional employee objects like `emp3`. By leveraging this function, one can streamline object creation, reduce errors, and improve code maintainability in scenarios involving multiple similar objects.

04 - JavaScript Constructors

Constructor functions in JavaScript simplify object creation by eliminating repetitive code, allowing developers to use the `new` keyword for efficient object initialization.

In the provided code below,

```
function createEmployeeObject(firstName, lastName, gender, designation) {
  var newObj = {};
  newObj.firstName = firstName;
  newObj.lastName = lastName;
  newObj.gender = gender;
  newObj.designation = designation;

  return newObj;
}
```

The process of creating and returning a new object (`var newObj = {}` and `return newObj`) is repetitive when writing multiple functions to create different types of objects. JavaScript simplifies this using a constructor function, which is called with the `new` keyword. Unlike other languages where the new keyword is used with a class name, in JavaScript, it is used with a function. The `new` keyword automates the creation of a new object and assigns it to `this`, making it available within the function. Behind the scenes, JavaScript essentially does the following:

```
function createEmployeeObject(firstName, lastName, gender, designation) {
  // var this = {};
  this.firstName = firstName;
  this.lastName = lastName;
  this.gender = gender;
  this.designation = designation;
  // return this;
}
```

The commented lines (`var this = {}` and `return this`) represent the implicit actions performed by JavaScript when a constructor function is invoked using new. This eliminates the need to explicitly create and return the object, streamlining object creation in the codebase.

Highlights

- 🔧 Constructor functions simplify object creation.
- 🏗️ Using the `new` keyword initializes a new object.
- 🔄 JavaScript automatically handles object creation logic.
- 📦 Developers can populate objects using the `this` keyword.
- ⚙️ Constructor functions eliminate boilerplate code.
- 💥 The pattern is common across various object types.
- 📚 JavaScript's approach differs from traditional OOP languages.

05 - Difference between regular functions and constructors

It explores two methods for creating objects in JavaScript: a standard function and a constructor function with the `new` keyword. The first method, `createBicycle`, uses a standard function to create objects manually. It initializes an empty object, assigns properties like `cadence`, `speed`, and `gear` from the provided arguments, and then returns the constructed object.

```
function createBicycle(cadence, speed, gear) {
  var newBicycle = {};
  newBicycle.cadence = cadence;
  newBicycle.speed = speed;
  newBicycle.gear = gear;
  return newBicycle;
}

var bicycle1 = createBicycle(50, 20, 4);
var bicycle2 = createBicycle(20, 5, 1);
```

We also have, `bicycleConstructor`, which simplifies object creation. By using the `new` keyword, JavaScript automatically initializes a new object, binds it to `this`, and returns it implicitly. The function defines properties directly on `this`, avoiding the need for explicit initialization and return statements.

```
function bicycleConstructor(cadence, speed, gear) {
  this.cadence = cadence;
  this.speed = speed;
  this.gear = gear;
}

var bicycle3 = new bicycleConstructor(50, 20, 4);
```

Although JavaScript lacks explicit markers for constructor functions, the naming convention of starting function names with a capital letter i.e. PascalCase (e.g., `BicycleConstructor`) serves as a visual cue for their intended use. Additionally, the segment compares JavaScript's constructors to the class-based syntax in other languages and hints at potential pitfalls when constructors are used improperly, setting the stage for future discussions.

💡 Note: If you try to call a constructor function without `new` keyword, it will not work.

06 - Switching function types and calls

It explains the differences between using constructor functions with the `new` keyword versus calling them as regular functions, emphasizing the importance of proper usage.

Constructor functions in JavaScript are specifically designed to be called with the `new` keyword, which automates the creation of a new object, binds it to `this`, and implicitly returns it. In contrast, regular functions can create objects without the need for `new`. However, using the `new` keyword with a regular function can lead to unnecessary code execution and inefficiencies.

When a constructor function is called without the `new` keyword, JavaScript does not automatically create or return an object, which often results in the function returning `undefined`. This happens because JavaScript defaults to returning `undefined` when no explicit return statement is provided. On execution of constructor function without `new` keyword, this keyword will refer to global window and the properties will be assigned to global window object. This behavior underscores the importance of correctly distinguishing between regular functions and constructor functions.

Mixing these function types without adhering to conventions, such as capitalizing constructor function names, can lead to unexpected behaviors, errors, and confusion in the codebase. Following established naming conventions and usage patterns helps ensure clarity, proper function usage, and maintainable code.

Unit 02 - Understanding the `this` reference

07 - Function Execution Types

Understanding function execution types in JavaScript is essential for grasping the `this` keyword. JavaScript functions can be executed in four distinct ways, which influence how the `this` context is determined. Understanding these methods is crucial for advanced JavaScript development.

• 🧠 **Direct Function Call**: Calling a function directly executes it in the global context or the local context if inside another function. This is the simplest form of function invocation.

• 🧠 **Method Invocation**: When a function is called as a property of an object, it executes in the context of that object, allowing access to its properties through `this`. This distinction is important for object-oriented programming in JavaScript.

• 🧠 **Constructor Invocation**: Using the `new` keyword creates a new object, and `this` within that function refers to the newly created object. This is fundamental for creating instances of objects.

```
function foo() {
  console.log("Hello");
}

foo(); // Method #1

var obj = {};

obj.foo = function() {
  console.log("Hello");
};

obj.foo(); // Method #2


new foo(); // Method #3

           // Method #4
```

08 - The this argument values

The execution context in JavaScript determines the value of this in function calls, varying by the method of invocation.

Highlights

- 🌎 Execution context defines how functions are called and their environment.
- 📚 `this` is an implicit argument in JavaScript function executions.
- 🔍 Different methods of function calls affect the value of `this`.
- 🦖 Calling a function directly sets `this` to the global object.
- 📦 Using an object method sets `this` to the object itself.
- ✨ The `new` keyword creates a new object, with `this` referring to that object.
- ⚠️ Method 4 will explore practical use cases for `this`.

Key Insights

🌐 Execution Context: Every function call in JavaScript occurs within a specific context that includes variables and scope information, essential for proper execution. Understanding this context is crucial for debugging and writing effective code.

🔗 The this Keyword: This keyword behaves differently based on how a function is called, making it a pivotal concept for JavaScript developers. Knowledge of this is vital for object-oriented programming in JavaScript.

🎯 Direct Function Calls: When a function is invoked directly (e.g., func()), this points to the global object, which highlights how context can vary widely between environments (browser vs. Node.js).

🏷️ Method Calls: When a function is called as a property of an object (e.g., obj.func()), this refers to that object. This showcases how object-oriented principles manifest in JavaScript.

🆕 Constructor Functions: Using the new keyword creates a new instance where this refers to the newly created object, illustrating JavaScript's prototypal inheritance model.

🔄 Method Variability: The value of this is predictable based on the function invocation method, making it easier to anticipate behavior and avoid bugs in code.

⚡ Next Steps: Understanding the fourth method of function calls and practical applications of this is essential for mastering JavaScript's execution context and resolving common issues related to scope.

09 - Working on objects with this reference

```
function Bicycle(cadence, speed, gear, tirePressure) {
  this.cadence = cadence;
  this.speed = speed;
  this.gear = gear;
  this.tirePressure = tirePressure;
  this.inflateTires = function () {
    this.tirePressure += 3;
  };
}

// Creating an instance of Bicycle
var bicycle1 = new Bicycle(50, 20, 4, 25);

// Calling the method to increase tire pressure
bicycle1.inflateTires();
```

Over here the `Bicycle` constructor function demonstrates how objects in JavaScript can have properties and methods that operate within the context of the object they belong to. The `inflateTires` method is a particularly important example because it highlights how the `this` keyword works in JavaScript and how it enables the modification of object-specific properties.

When a new `Bicycle` instance is created using the `Bicycle` constructor function, the `this` keyword inside the function refers to the newly created object. For example, when `bicycle1` is instantiated using the `new Bicycle(50, 20, 4, 25)` call, `this` in the constructor points to `bicycle1`. The constructor assigns the passed values to the object's properties (`cadence`, `speed`, `gear`, and `tirePressure`) and defines the `inflateTires` method directly on the object.

The `inflateTires` method uses `this.tirePressure` to access and modify the `tirePressure` property of the object it is called on. When the method is invoked on an instance, such as `bicycle1.inflateTires()`, the `this` keyword inside the method dynamically refers to the `bicycle1` instance. This is a crucial feature of JavaScript's `this` behavior: its value is determined by the object that calls the method.

```
var bicycle1 = new Bicycle(50, 20, 4, 25);
console.log(bicycle1.tirePressure); // Output: 25

bicycle1.inflateTires();
console.log(bicycle1.tirePressure); // Output: 28
```

Here, calling `bicycle1.inflateTires()` increases `bicycle1`'s `tirePressure` by 3, because the `this` keyword inside the method specifically refers to the `bicycle1` instance. If the method were called on a different object (e.g., another `Bicycle` instance), `this` would refer to that object, and only its `tirePressure` would be updated.

This behavior demonstrates how JavaScript allows methods to operate within the scope of the object they belong to, providing a powerful way to manage object-specific data. By using the `this` keyword, methods like `inflateTires` can dynamically adapt to the context of the object they are called on, ensuring that changes are made only to the relevant instance. This encapsulation of behavior and data is fundamental to object-oriented programming in JavaScript.

10 - Using the call function

In the video, the concept of function context in JavaScript is explored through a practical example of creating a `Mechanic` object that borrows the `inflateTires` method from a `Bicycle` object. This example emphasizes the importance of understanding how the `this` keyword behaves in different contexts and how to manage it effectively when reusing methods across objects.

Code Explanation:

1. **Bicycle Constructor**: A `Bicycle` constructor function is defined, allowing the creation of bicycle objects with properties like `cadence`, `speed`, `gear`, and `tirePressure`. Each `Bicycle` instance has an `inflateTires` method that increases the `tirePressure` by 3, using `this.tirePressure` to refer to the specific object it is called on.
2. **Mechanic Constructor**: A `Mechanic` constructor function is used to create mechanic objects with a `name` property. In this example, a mechanic named "Mike" is created.
3. **Method Borrowing**: The `inflateTires` method from the `bicycle1` object is assigned to the `Mechanic` instance `mike`. However, when `mike.inflateTires()` is invoked, the `this` keyword inside the method refers to the `mike` object, which does not have a `tirePressure` property. This results in an error or an invalid operation (e.g., `this.tirePressure += 3` will produce `NaN` because `undefined + 3` is not a valid operation).

Fixing the Context Issue:

To address this, we need to ensure that the `this` keyword inside the `inflateTires` method refers to the correct object (e.g., a `Bicycle` instance). This can be done in two ways:

1. **Passing the Bicycle Object as an Argument**: Modify `inflateTires` to accept an object as an argument and directly update its `tirePressure`.
2. **Binding the Context Explicitly**: Use the `call` method to explicitly set the value of `this` to the `bicycle1` object when invoking `inflateTires`.

Here's the final code that demonstrates both approaches:

Code:

```
// Bicycle constructor
function Bicycle(cadence, speed, gear, tirePressure) {
  this.cadence = cadence;
  this.speed = speed;
  this.gear = gear;
  this.tirePressure = tirePressure;
  this.inflateTires = function () {
    this.tirePressure += 3; // This assumes 'this' points to a Bicycle object.
  };
}

// Mechanic constructor
function Mechanic(name) {
```

```
  this.name = name;
}

// Creating objects
var bicycle1 = new Bicycle(50, 20, 4, 25);
var mike = new Mechanic("Mike");

// Borrowing the inflateTires method
mike.inflateTires = bicycle1.inflateTires;

// Fix 1: Pass the Bicycle object as an argument
mike.inflateTires = function (bicycle) {
  bicycle.tirePressure += 3;
};
mike.inflateTires(bicycle1);
console.log(bicycle1.tirePressure); // Output: 28

// Fix 2: Use call to explicitly bind the context
mike.inflateTires = bicycle1.inflateTires; // Reassign the original method
mike.inflateTires.call(bicycle1);
console.log(bicycle1.tirePressure); // Output: 31
```

Key Insights:

- **Understanding `this`**: The `this` keyword refers to the calling object, not the object where the method is defined. This can lead to errors when borrowing methods.
- **Managing Context**: Using `call`, `apply`, or `bind` allows you to explicitly set the value of `this`, ensuring that methods behave as expected.
- **Modular Functions**: Methods like `inflateTires` can be adapted for reuse by passing the required object as an argument or binding the correct context at runtime.
- **Error Prevention**: Without proper management of `this`, operations can lead to invalid results, such as attempting to modify properties that do not exist on the calling object.

This example underscores the flexibility and challenges of working with `this` in JavaScript, encouraging developers to pay close attention to context when reusing or borrowing methods.

Unit 03 - Prototypes

---

11 - When constructors aren't good enough

- **Prototypes** in JavaScript allow you to create objects based on a shared **template** or **blueprint**.
- Unlike class-based programming languages (e.g., Java, C++), JavaScript doesn't have classes (at least before ES6). Instead, it uses prototypes to define reusable behaviors across objects.
- While prototypes aren't exactly like classes, they serve a similar purpose by enabling objects to share behaviors without duplicating them.

Comparison with Class-Based Languages

- In languages like Java or C++, objects are **instances of classes**, and these classes act as the blueprint for the objects.
- In such languages:
  - You cannot create objects "out of thin air." Every object must be an instance of a class.

    - Methods (functions) are shared across all instances of a class. The methods are defined once in the class and reused by all instances, saving memory.
- In contrast, JavaScript:
    - Does not enforce class-based object creation. Objects can be created independently without needing a "class."
    - Does not inherently distinguish between "properties" and "methods." In JavaScript:
        - Objects have properties, which can store values (primitive types, objects, or functions).
        - A function property may behave like a "method," but it is not inherently tied to a class or the object in a traditional sense.

The Problem with Constructor Functions

- JavaScript uses **constructor functions** to create objects with shared properties and methods.

```
function Bicycle(cadence, speed, gear, tirePressure) {
  this.cadence = cadence;
  this.speed = speed;
  this.gear = gear;
  this.tirePressure = tirePressure;
  this.inflateTires = function () {
    this.tirePressure += 3;
  };
}
// When you create an instance using new Bicycle, like:
var bicycle1 = new Bicycle(50, 20, 4, 25);
var bicycle2 = new Bicycle(40, 15, 5, 30);
```

  Each object (`bicycle1` and `bicycle2`) will have:
    - Its own copies of properties (`cadence`, `speed`, etc.).
    - A **new copy of the `inflateTires` function** for each instance. For every new `Bicycle` created, a **new function object** is created and stored in memory.

Drawback:

- If you have a large number of objects (e.g., 1,000 employees in an employee management system), each object will unnecessarily have its own copy of methods, leading to memory inefficiency.
- This is wasteful because the logic of methods (like `inflateTires`) remains the same for all instances.

This is where Prototypes comes as a solution.

Note: There is a new `class` keyword in the newer version of JavaScript(ES6) that simulates class-like behaviour, but JavaScript does not have the class concept.

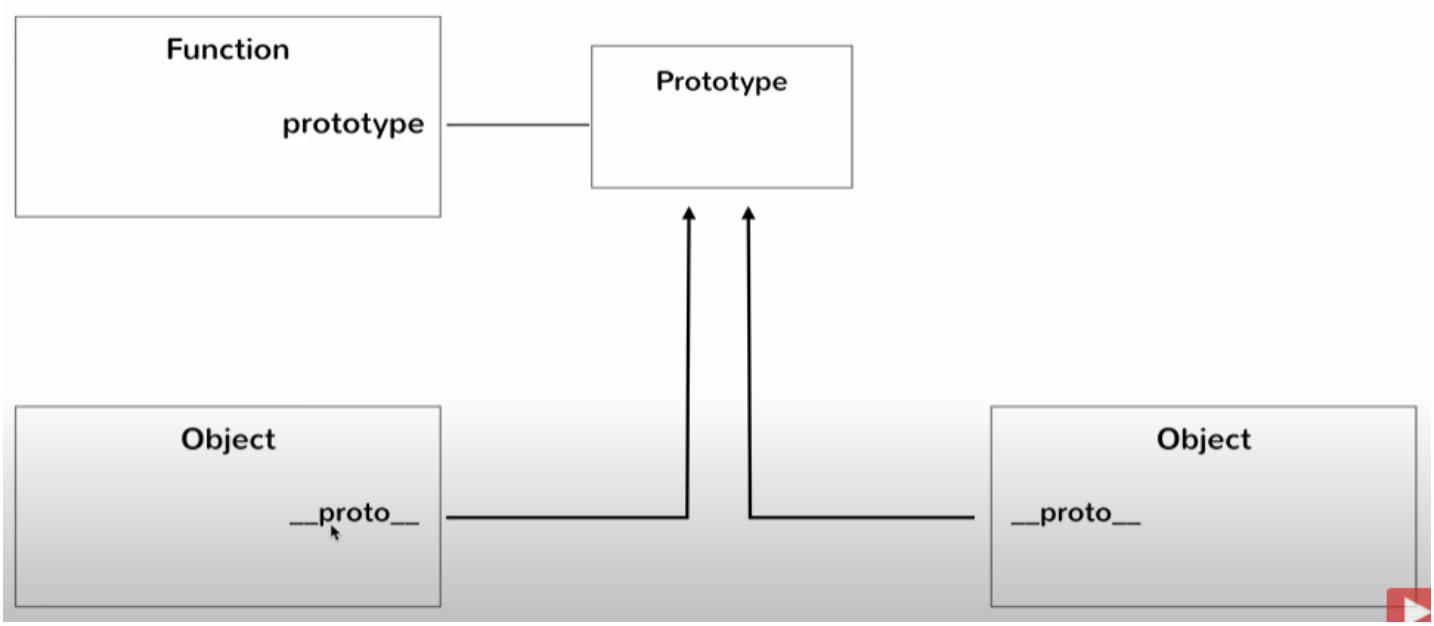
12 - Introducing the prototype

1. **Functions in JavaScript Are Objects**:
    - Every function in JavaScript is an object.
    - When a function (`function foo() {}`) is defined, the JavaScript engine creates two objects:
        1. The function object itself (e.g., `foo`).
        2. A prototype object associated with the function.
2. **Accessing Function Objects and Prototype Objects**:

- The function object can be accessed directly using the function name (e.g., `foo`).
- The prototype object is accessible through the `prototype` property of the function (e.g., `foo.prototype`).

3. **Prototype Object Creation**:
   - The JavaScript engine creates a prototype object for every function, even if the function does nothing (e.g., empty functions like `function foo() {}`).
   - This prototype object is associated with the function via its `prototype` property.

4. **Using the `new` Keyword**:
   - When a function is called with the `new` keyword, the JavaScript engine:
     1. Creates a new object.
     2. Executes the function, setting `this` to the new object.
     3. Links the new object to the prototype object of the function.
   - The resulting object has a special property, `__proto__`, which points to the function's prototype object.



5. **The Role of `__proto__`**:
   - The `__proto__` property is automatically added to any object created using the `new` keyword.
   - It links the new object to the prototype object of the function.
   - All objects created using the same function share the same prototype object.

6. **Behavior of the Prototype Object**:
   - If the function is called without `new`, the prototype object is not used.
   - If the function is called with `new`, the new object references the prototype object via `__proto__`.

7. **Key Observations**:
   - Functions that do not involve object creation still have a prototype object, but it is unused unless the `new` keyword is used.
   - Objects created using the `new` keyword share a single prototype object, ensuring efficient memory usage.

These steps are fundamental for understanding how JavaScript utilizes prototypes to manage object behavior and inheritance. The significance of the prototype object and the `__proto__` property will become clearer in

subsequent lessons.

13 - Property lookup with prototypes

- **Prototype Validation**:
  - When a function (e.g., `function Foo`) is created, its `.prototype` property refers to the prototype object associated with the function.
  - Objects created using the `new` keyword (e.g., `let obj = new Foo()`) contain a special `__proto__` property that points to the function's `.prototype` object.
  - You can validate this by setting a property on the `.prototype` object (e.g., `Foo.prototype.test = "prototype property"`), which becomes accessible via `obj.__proto__.test` or `Foo.prototype.test`.

```
// Create a constructor function
function Foo() {}

// Prototype object of Foo
console.log(Foo.prototype); // {}

const obj = new Foo();

// Validate the prototype link
console.log(obj.__proto__ === Foo.prototype); // true

// Set a property on the prototype
Foo.prototype.test = "This is a prototype property";

// Access the prototype property from the object
console.log(Foo.prototype.test); // "This is a prototype property"
console.log(obj.__proto__.test); // "This is a prototype property"
console.log(obj.test); // "This is a prototype property"
```

- **Property Lookup Process**:
  - When accessing a property on an object (e.g., `obj.property`), the JavaScript engine:
    1. Checks the object itself for the property.
    2. If the property is not found, it checks the `__proto__` (prototype object).
    3. If the property is still not found, it continues up the prototype chain (if applicable).
    4. If no match is found in the entire chain, it returns `undefined`.
- **Prototype Chain Behavior**:
  - If a property exists on the object itself, the prototype object is not consulted. For example:
    - Setting `obj.test = 10` overrides the prototype's `test` property.
    - Accessing `obj.test` returns `10`, not the value from the prototype.
  - Deleting the property on the object (e.g., `delete obj.test`) re-enables access to the prototype's `test` property.
- **Practical Example**:
  - Suppose `Foo.prototype.hello = "Hello from prototype"`:
    - Accessing `obj.hello` will return the value from the prototype since `obj` itself does not have a `hello` property.

    - Adding `obj.hello = "Hello from object"`:
        - Now, `obj.hello` returns `"Hello from object"`, and the prototype's `hello` property is ignored.

- This lookup mechanism is implicit and transparent, making it difficult to tell if a property is from the object or the prototype without explicitly examining the object.
- To check explicitly:

```
// Check if the property exists on the object itself
console.log(obj2.hasOwnProperty("test")); // false (if it comes from the prototype)

// Check the prototype chain for the property
console.log("test" in obj2); // true
```

Why This Mechanism Exists?

This allows JavaScript to create **shared behavior** across objects (like a blueprint or template). Instead of duplicating methods and properties for every instance, shared behaviors reside in the prototype. More on this will be explored in future lessons!

14 - Object behaviors using prototypes

Why Prototype Lookup Exists

- Prototype lookup enables **shared behavior** across multiple objects created from the same constructor function.
- Objects created using a constructor share the same prototype.
- This avoids duplication of properties/methods for each object, saving memory.

Prototype in Constructor Functions

Objects created using the `new` keyword inherit from the constructor's `.prototype`. Example: Constructor with a Prototype

```
function Employee(name) {
  this.name = name; // Instance-specific property
}

// Adding shared behavior to the prototype
Employee.prototype.playPranks = function () {
  console.log("Prank played!");
};

// Create objects
const emp1 = new Employee("Jim");
const emp2 = new Employee("Pam");

// Access shared method
emp1.playPranks(); // "Prank played!"
emp2.playPranks(); // "Prank played!"

// Check property ownership
console.log(emp1.hasOwnProperty("playPranks")); // false (comes from prototype)
```

- Adding methods directly to the constructor (`this.method = function()`) creates a new copy of the method for every object. Instead, adding methods to the prototype ensures only one shared copy is created.

```
// Inefficient Method Definition:
function Employee(name) {
  this.name = name;
  this.playPranks = function () {
    console.log("Prank played!");
  };
}

const emp1 = new Employee("Jim");
const emp2 = new Employee("Pam");

console.log(emp1.playPranks === emp2.playPranks); // false (different copies)

----------------------------------------------------------------
// Efficient Method Definition:
function Employee(name) {
  this.name = name;
}
Employee.prototype.playPranks = function () {
  console.log("Prank played!");
};

const emp1 = new Employee("Jim");
const emp2 = new Employee("Pam");

console.log(emp1.playPranks === emp2.playPranks); // true (shared method)
```

Dynamic Prototype Modifications

- Prototype properties/methods can be added **at runtime**, and all existing objects will immediately inherit them.

```
// Add a method to the prototype after object creation
Employee.prototype.greet = function () {
  console.log(`Hello, ${this.name}!`);
};

emp1.greet(); // "Hello, Jim!"
emp2.greet(); // "Hello, Pam!"
```

Key Characteristics of Prototype Behavior

1. **Dynamic Runtime Lookup**: Prototype properties are checked at runtime, so changes to the prototype are immediately reflected on all objects.
2. **Shared Behavior**: Shared methods reduce memory usage and simplify updates.
3. **No Need for Upfront Definition**: Unlike class-based languages, prototype methods can be added dynamically after object creation.

4. In traditional class-based languages, all behaviors must be defined upfront before object creation.

15 - Object Links With Prototypes

Note: The double underscores are referred to as “dunder” as in “Dunder Mifflin”. So this property is called “dunder-proto”.

In JavaScript, objects and functions are connected via a network of prototype relationships that allow behavior sharing and object creation. Here's how it works:

- When a **function** is created, it gets a special property called `prototype` that points to a **prototype object**:

```
function Foo() {}
console.log(Foo.prototype); // Prototype object
```

- When an object is created using the `new` keyword, the object gets a special property `__proto__` (also called "Dunder Proto"), which links it to the function's prototype:

```
const a = new Foo();
console.log(a.__proto__ === Foo.prototype); // true
```

- The prototype object itself has a `constructor` property that points back to the function:

```
console.log(Foo.prototype.constructor === Foo); // true
console.log(a.__proto__.constructor === Foo); // true
```

This setup allows an object to "inherit" behaviors defined on the prototype:

```
Foo.prototype.greet = function () {
  console.log("Hello!");
};

a.greet(); // "Hello!"
// If the object (a) doesn’t have a property or method, JavaScript looks up the chain to
__proto__ (i.e., the prototype) to find it.
```

You can even use the `constructor` to create new objects:

```
const b = new a.__proto__.constructor();
console.log(b instanceof Foo); // true
// While this works, it’s not recommended because modifying __proto__ or constructor can
lead to unexpected behavior:
a.__proto__.constructor = function Bar() {};
console.log(a.__proto__.constructor); // Bar
// Changing constructor breaks the relationship between the object and its original
creator.
```

Best Practices

Instead of relying on `__proto__`, use the following:

- `Object.getPrototypeOf(obj)` to access the prototype:

```
console.log(Object.getPrototypeOf(a) === Foo.prototype); // true
```

- `Object.setPrototypeOf(obj, prototype)` to modify the prototype safely.

When defining shared behavior, always use the constructor's `prototype` property:

```
Foo.prototype.sayHello = function () {
  console.log("Hello from Foo!");
};

const c = new Foo();
c.sayHello(); // "Hello from Foo!"
```

16 - The Object function

The `Object` Function in JavaScript

The `Object` function in JavaScript is both a global function and an object. It acts as a global constructor function that allows you to create objects. For example:

```
// Simplest way to create an object
let simple = {};

// Another way to create an object using the Object function
let obj = new Object();
```

Both approaches are equivalent. `{}` is simply a shorthand for `new Object()`. To prove this, you can check the prototype chain:

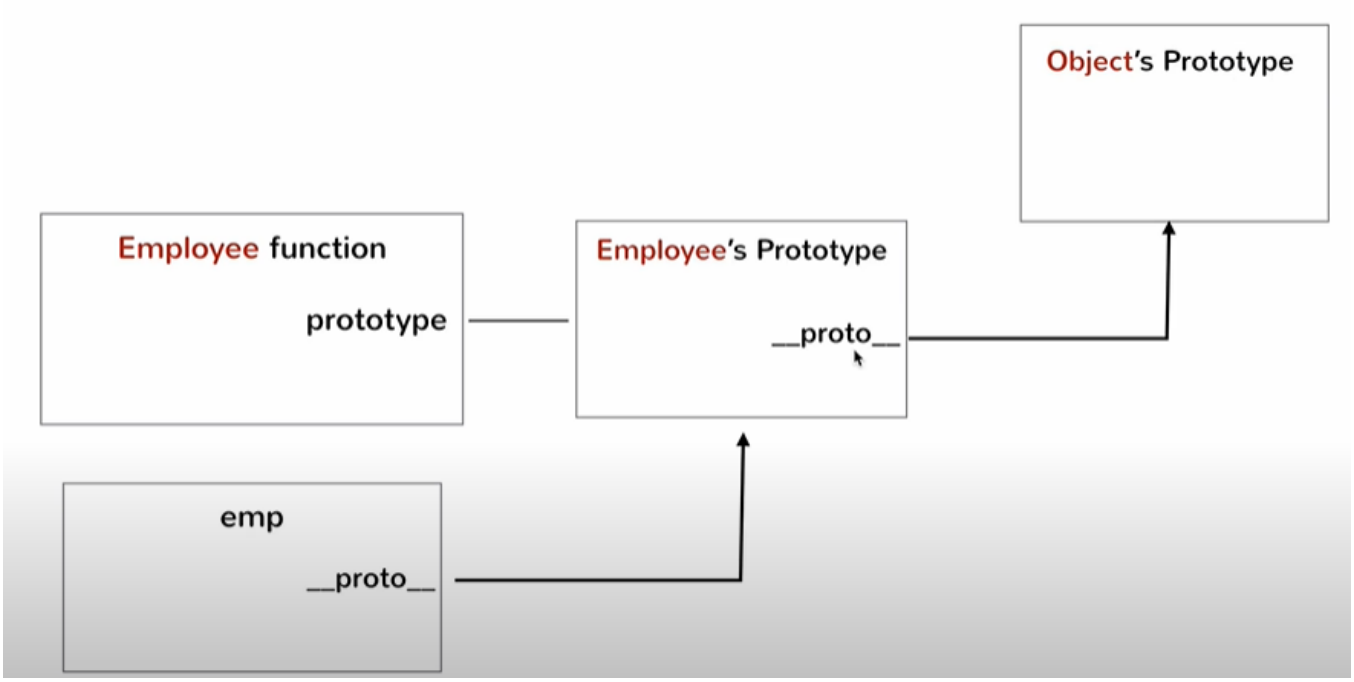
```
console.log(simple.__proto__ === obj.__proto__); // true
```

When you create an object using `{}`, JavaScript internally calls `new Object()` behind the scenes.

17 - The Prototype object

- When we create an object using a constructor function (e.g., `new Employee()`), the following happens:
    1. The `Employee` function itself is created as a function object.
    2. A `prototype` object is automatically created for the `Employee` function.
    3. The `__proto__` property of the created object (e.g., `emp`) points to the `Employee`'s `prototype`.

4. The `prototype` object for a constructor (e.g., `Employee.prototype`) is itself created by calling `new Object()`. This means the `__proto__` of `Employee.prototype` points to `Object.prototype`.




```
console.log(Employee.prototype.__proto__ === Object.prototype); // true
```

5. Properties can be added to prototypes at any level, making them accessible to all instances:

- Adding to the instance's prototype (`Employee.prototype`):

```
Employee.prototype.parentProp = "Parent of Employee";
console.log(emp.parentProp); // "Parent of Employee" // because of Prototype
Chain
```

- Adding to `Object.prototype`:

```
Object.prototype.globalProp = "Grandparent Property";
console.log(emp.globalProp); // "Grandparent Property"
```

- Adding properties to `Object.prototype` affects **all objects** in JavaScript because every object's prototype chain ends at `Object.prototype`.
- **Caution**: This is similar to using global variables and should be avoided in large systems to prevent conflicts.
- The `Object.prototype` itself has a `__proto__` that points to `null`.
- This is the end of the prototype chain, preventing infinite loops during property lookups.

```
console.log(Object.prototype.__proto__); // null
// Instance → Constructor's Prototype → Object.prototype → null.
```

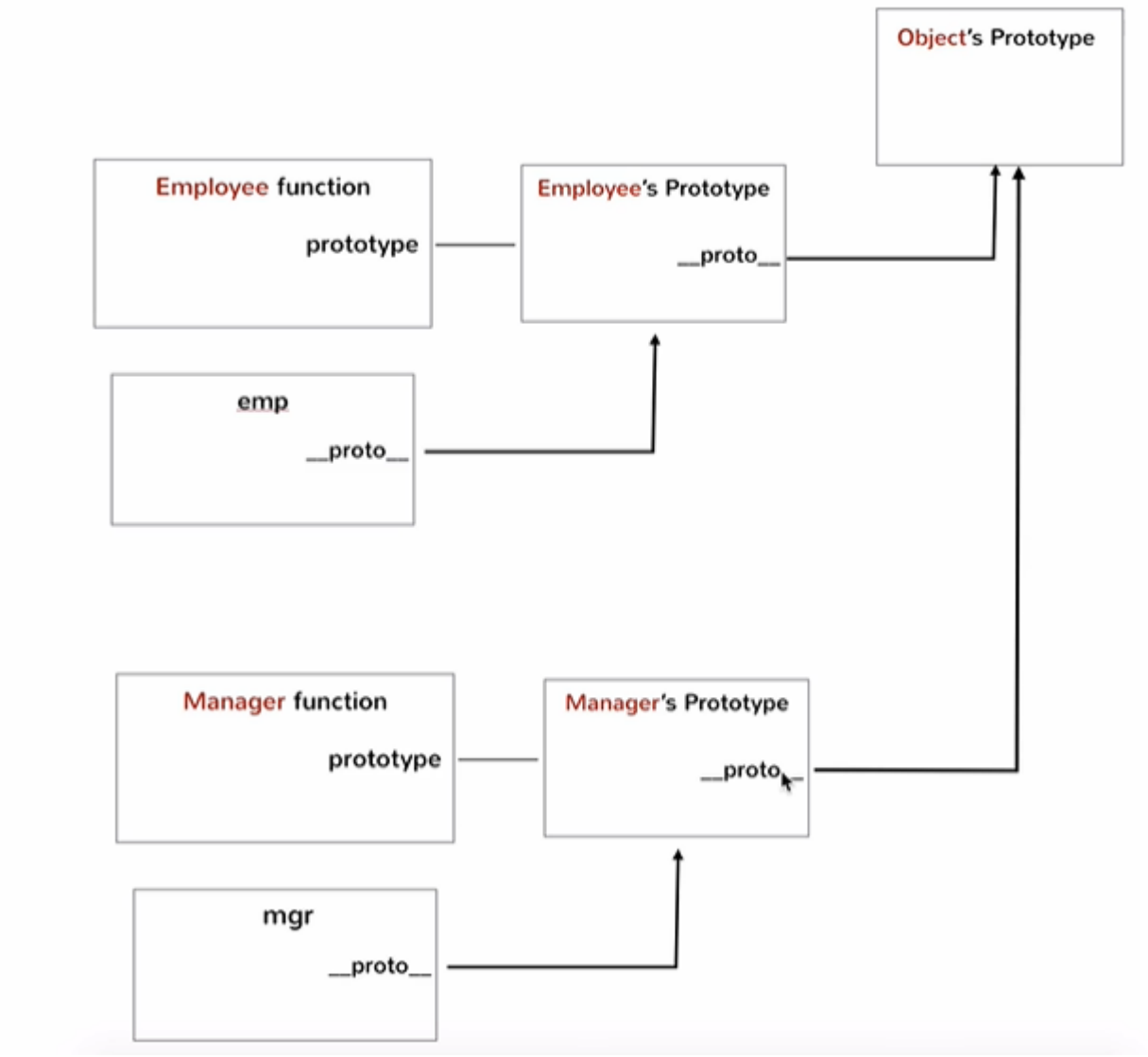
18 - Inheritance In JavaScript

In JavaScript, inheritance is achieved via the **prototype chain**, allowing objects to inherit properties and methods from other objects. This example demonstrates how to implement multi-level inheritance using JavaScript's prototype system.

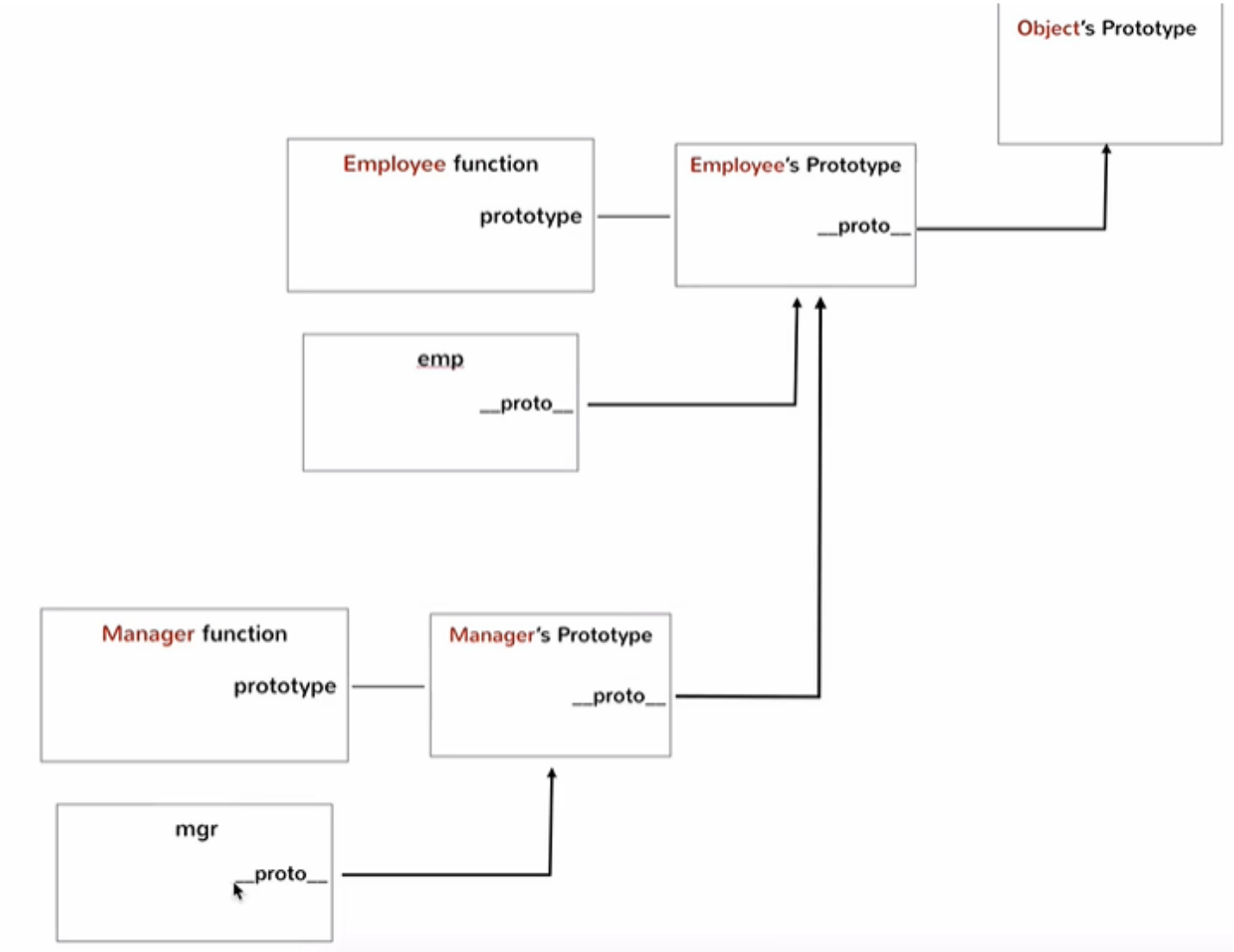
- **Prototype Chain**: Every function in JavaScript has a `prototype` property, and instances created using that function inherit methods and properties from the function's prototype.
- **Setting the Prototype**: By modifying the `__proto__` property, we can change the prototype chain to share behaviors between different constructors (e.g., `Employee` and `Manager`).

```
// Employee constructor function
function Employee(name) {
  this.name = name;
}

// Add method to Employee prototype
Employee.prototype.getName = function () {
  return this.name;
};

// Create an instance of Employee
let emp1 = new Employee("John");
console.log(emp1.getName()); // Output: John

// Manager constructor function, inherits from Employee
function Manager(name, department) {
  this.name = name;
  this.department = department;
}

// Add Manager-specific method to Manager prototype
Manager.prototype.getDepartment = function () {
  return this.department;
};

// Create an instance of Manager
let mgr1 = new Manager("Michael", "Sales");

console.log(mgr1.getName()); // Output: Type Error
console.log(mgr1.getDepartment()); // Output: Sales (Specific to Manager)

// Let's make Manager inherit properties of Employee
mgr1.__proto__.__proto__ = Employee.prototype;
console.log(mgr1.getName()); // Output: Michael
```

- Prototype chain before linking Employee and Manager:



- Prototype chain after linking Employee and Manager:



This concept is the foundation of object-oriented programming in JavaScript and can be extended further to create deep inheritance hierarchies.

Thank you!