(10.0.2) ES6 Intermediate - Constructor - Object.create()

Understanding Object.create and Prototypes in JavaScript

Objects are at the heart of JavaScript, serving as the fundamental building blocks for nearly every aspect of the language. Whether you're new to JavaScript or a seasoned developer, grasping the concept of objects and how they relate to prototypes is essential. In this article, we'll start with the basics and gradually delve into more advanced concepts.

The Foundation: Objects in JavaScript

At the core of JavaScript lies the concept of objects, which consist of key-value pairs. You're likely familiar with creating objects using curly braces `{}` and adding properties and methods to them using dot notation. Let's revisit this basic concept with an example:

```
let animal = {}
animal.name = 'Leo'
animal.energy = 10

animal.eat = function (amount) {
console.log(`${this.name} is eating.`)
this.energy += amount
}

animal.sleep = function (length) {
console.log(`${this.name} is sleeping.`)
this.energy += length
}

animal.play = function (length) {
console.log(`${this.name} is playing.`)
this.energy -= length
}
```

In this example, we create an `animal` object and add properties like `name` and `energy`, along with methods such as `eat`, `sleep`, and `play`. While this approach works, it has some limitations, especially when you need to create multiple instances with similar methods.

Functional Instantiation: A Step Forward

To address the issue of redefining methods for each instance, we can encapsulate object creation in a constructor function. This pattern is known as Functional Instantiation. Here's an example:

```
function Animal (name, energy) {
let animal = {}
animal.name = name
animal.energy = energy

animal.eat = function (amount) {
console.log(`${this.name} is eating.`)
this.energy += amount
}

animal.sleep = function (length) {
console.log(`${this.name} is sleeping.`)
this.energy += length
}

animal.play = function (length) {
console.log(`${this.name} is playing.`)
this.energy -= length
}

return animal
}

const leo = Animal('Leo', 7)
const snoop = Animal('Snoop', 10)
```

Now, when we want to create new instances of animals, we simply invoke the `Animal` function with the respective name and energy level. However, this approach still has room for improvement, particularly in terms of memory efficiency.

Shared Methods: A More Efficient Approach

In the previous example, each instance of an animal includes its own copies of the `eat`, `sleep`, and `play` methods. This redundancy leads to memory wastage. To address this, we can create a separate object to hold shared methods and reference it from each instance. We'll call this pattern Functional Instantiation with Shared Methods:

```
const animalMethods = {
eat(amount) {
console.log(`${this.name} is eating.`)
this.energy += amount
},
sleep(length) {
console.log(`${this.name} is sleeping.`)
this.energy += length
},
play(length) {
console.log(`${this.name} is playing.`)
this.energy -= length
}
}

function Animal (name, energy) {
let animal = {}
animal.name = name
animal.energy = energy
animal.eat = animalMethods.eat
animal.sleep = animalMethods.sleep
animal.play = animalMethods.play

return animal
}

const leo = Animal('Leo', 7)
const snoop = Animal('Snoop', 10)
```

By moving the shared methods to the `animalMethods` object and referencing it from each instance, we eliminate memory redundancy and create more efficient objects. However, there's still a more elegant solution to this problem: using `Object.create`.

Object.create: A Smarter Way to Share Methods

`Object.create` is a powerful feature in JavaScript that allows you to create objects that delegate property lookups to another object. In simpler terms, if a

property is not found on an object, JavaScript will consult another object to see if it has that property. Let's explore this with an example:

```
const parent = {
name: 'Stacey',
age: 35,
heritage: 'Irish'
}

const child = Object.create(parent)
child.name = 'Ryan'
child.age = 7

console.log(child.name) // Output: Ryan
console.log(child.age) // Output: 7
console.log(child.heritage) // Output: Irish
```

In this example, the `child` object delegates property lookups to its `parent` object. When a property like `heritage` is not found on `child`, it is successfully retrieved from `parent`.

Prototypal Instantiation: The Elegant Approach

Now, let's enhance our previous animal example by utilizing `Object.create` to create more efficient objects. Instead of manually adding shared methods to each instance, we'll delegate property lookups to a shared methods object using `Object.create`. We'll call this pattern Prototypal Instantiation:

```
const animalMethods = {
eat(amount) {
console.log(`${this.name} is eating.`)
this.energy += amount
},
sleep(length) {
console.log(`${this.name} is sleeping.`)
this.energy += length
},
play(length) {
console.log(`${this.name} is playing.`)
this.energy -= length
}
}

function Animal (name, energy) {
```

```
let animal = Object.create(animalMethods)
animal.name = name
animal.energy = energy

return animal
}

const leo = Animal('Leo', 7)
const snoop = Animal('Snoop', 10)

leo.eat(10)
snoop.play(5)
```

With this approach, when you call methods like `leo.eat`, JavaScript will automatically delegate the method lookup to `animalMethods`. This not only simplifies your code but also ensures efficient memory usage.

Conclusion

Understanding the relationship between objects and prototypes is essential in JavaScript. While Functional Instantiation has its merits, Prototypal Instantiation with `Object.create` offers a more elegant and memory-efficient way to share methods among objects. As you continue to explore JavaScript, you'll find that mastering the use of prototypes can lead to cleaner, more maintainable, and efficient code.