Lesson:

Prototypes in JavaScript

PW SKILLS

Topics to be covered in

1. Introduction to prototype
2. Prototype Property, __proto__ & their difference.
3. Prototypical inheritance
4. Adding and accessing properties and methods to the prototype
5. Polymorphism and prototype chains

Introduction to prototype

In javascript, prototype property is basically an object which is also known as the prototype object, which can attach methods and properties in a prototype object which enables all the other objects to inherit these methods and properties.
In simple words, we can say that a prototype is a mechanism by which JavaScript objects inherit features from one another.

In javascript, whenever we create a new function the javascript engine adds a prototype property inside a function that has all the properties and methods which can be accessed and used by that particular function. We can also add more properties and methods to this prototype.

Let's see how a prototype looks before moving ahead -

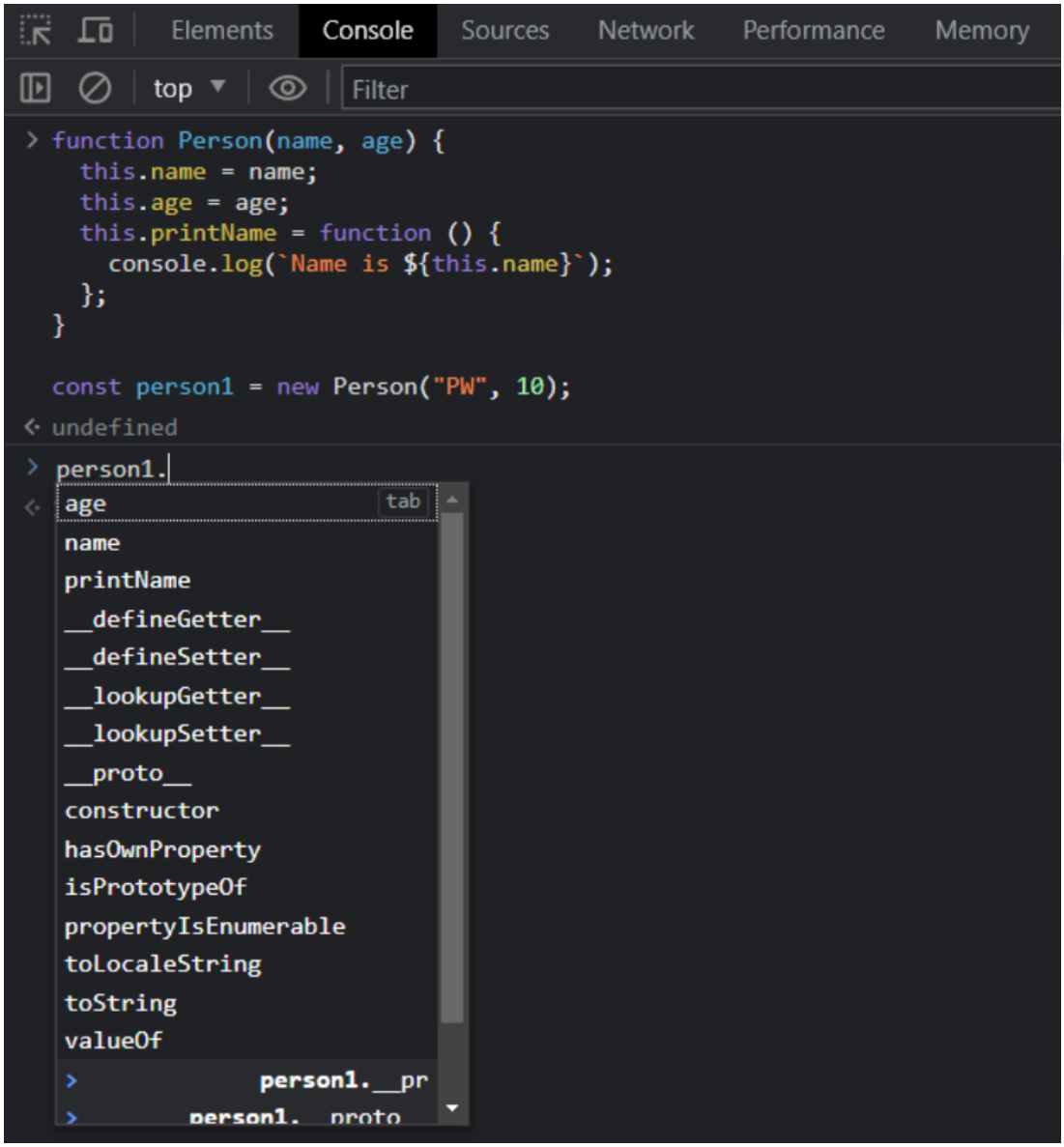
1. Example: Checking prototype

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.printDetails = function () {
    console.log(this.name + " " + this.age);
  };
}
console.log(Person.prototype);
```

In the above example, we created a simple object constructor in the browser console, as we had previously discussed the window object is not being displayed in the VS Code as there is a Node JS runtime environment that does not have access to the browser window. When we have logged our prototype we can see that our function has a prototype object attached to it.

2. Example: Checking methods and properties

Now if we try to use the dot operator after creating an instance of our object constructor to access its property and methods then we will be able to see a lot of properties and methods which is not even defined by us as we had only defined the name, age and printDetails
So the question is this: from where do these properties and methods come, in javascript every object has a built-in property, which is called prototype which is an object in itself which means that these prototypes can also have their own prototype
which we will discuss later in the class.



Important Note

Always keep in mind that whenever we look for a property or method it is first checked in the defined object constructor if not found then only it will check for the prototype.

Difference between prototype and __proto__

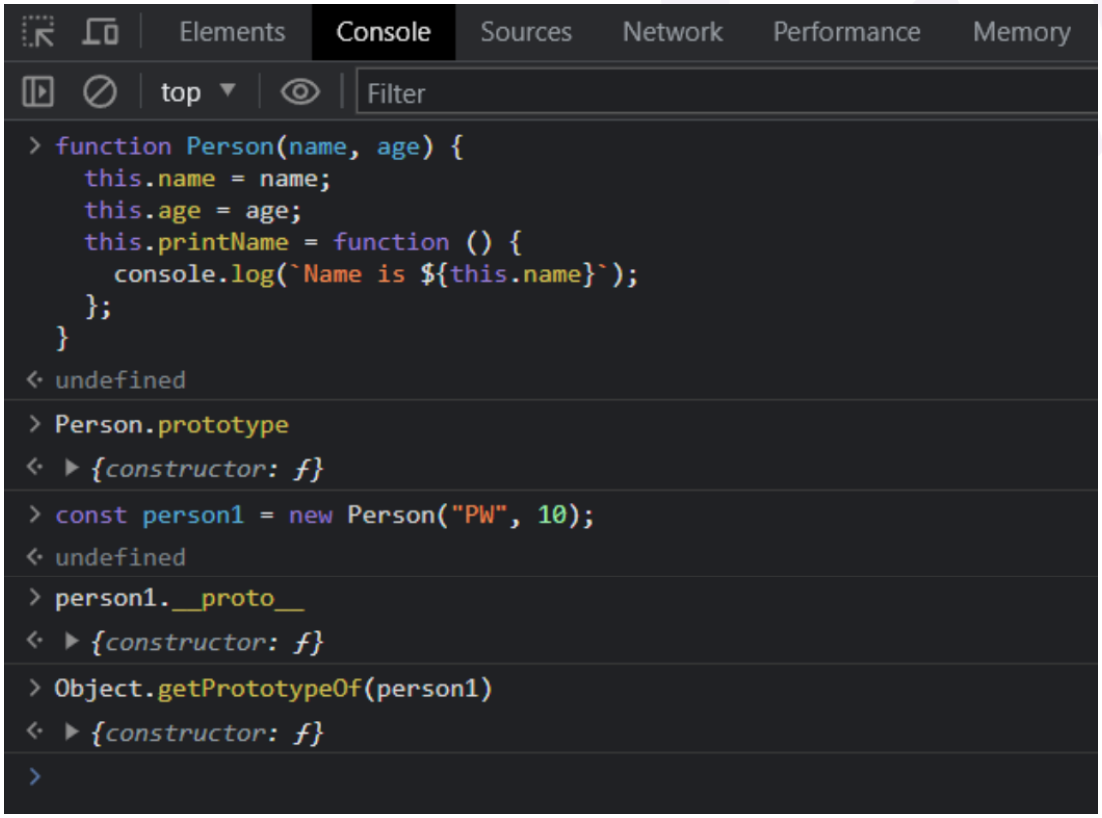
Prototype property

It is used in the constructor functions to define properties and methods for instances, which are located in the constructor function itself. We will see how we can add properties and methods using it.

__proto__ property

It is used to directly access the prototype of an instance, and it is located in the instances created using the constructor function.
Now it is not advisable to use __proto__ because of the standardization concerns, so instead of using this, it is advisable to use the Object.getPrototypeOf().

Example



In the above example, we can clearly see that we are using prototype property on the function constructor and __proto__ or Object.getPrototypeOf() for accessing the prototype of the instances created using the function constructor.

Prototypical inheritance

Prototypal inheritance is a fundamental concept in JavaScript that allows objects to inherit properties and methods from other objects. Every object in JavaScript has a prototype, which serves as a blueprint for creating new objects. When you access a property or method on an object, JavaScript looks up the prototype chain to find the property or method if it doesn't exist on the object itself.

1. Example using __proto__

```
const institute = {
  instituteName: "PW Skills",
  instituteLocation: "Banglore",
  instituteDomain: "Edtech",
};

const student = {
  studentName: "Vinay Pratap Singh",
  courseName: "Full Stack Web Development",
};

student.__proto___ = institute;
console.log(student.__proto___.instituteName);

// output
// PW Skills
```

In the above example, we are using the __proto__ to extend the prototype of the student with the institute one and are able to access it using the same __proto__.Similarly, we can use any number of nesting as per our need

Note: Using the __proto__ is not a recommended way, so try to avoid using it in your code.

2. Example using Object.create()

```
const institute = {
  instituteName: "PW Skills",
  instituteLocation: "Banglore",
  instituteDomain: "Edtech",
};
```

```
const student = Object.create(institute);
student.studentName = "Vinay Pratap Singh";
student.courseName = "Full Stack Web Development";
console.log(student);
console.log(student.instituteName);
console.log(student.courseName);
// output
// {
//   studentName: 'Vinay Pratap Singh',
//   courseName: 'Full Stack Web Development'
// }
// PW Skills
// Full Stack Web Development
```

In the above code we used the Object.create() method to create our student object which has inherited the institute prototype.

In the first console, we can see that we are able to see our student object which contains the studentName and studentBatch in it. But when we tried to access the instituteName then it is not available in the prototype of the student so it further down-searched in the prototype inherited by it and once it found it there it returns it, in case it was not there then it will search further down and if not available will return us the undefined value.

3. Example

In the above example, we had seen that if we have a property in the current object then it will not look into the prototype and will return it as a value.

So let's see this by using the example -

```
const institute = {
  instituteName: "PW Skills",
  instituteLocation: "Banglore",
  instituteDomain: "Edtech",
};

const student = Object.create(institute);
student.studentName = "Vinay Pratap Singh";
student.courseName = "Full Stack Web Development";
student.instituteName = "iNeuron";
console.log(student.instituteName);

// output
// iNeuron
```

Adding and accessing properties and methods to the prototype

We can add properties and methods to the prototype to extend its functionality.

1. Example

```
// function constructor
function Company(CompanyName, CompanyLocation, CompanyDomain)
{
  this.CompanyName = CompanyName;
  this.CompanyLocation = CompanyLocation;
  this.CompanyDomain = CompanyDomain;
}

const company1 = new Company("PW Skills", "Banglore",
"Edtech");
console.log(company1);

// adding the method to prototype
Company.prototype.printCompanyDetails = function () {
  console.log(
    `${this.CompanyName} is located in
${this.CompanyLocation} and it is an ${this.CompanyDomain}
company`
  );
};

company1.printCompanyDetails();

// output
// Company {
//   CompanyName: 'PW Skills',
//   CompanyLocation: 'Banglore',
//   CompanyDomain: 'Edtech'
// }
// PW Skills is located in Banglore and it is an Edtech
company
```

Note: We cannot directly add any method or property in a constructor function, if we want then have to use a prototype to do so as demonstrated in the above example.

2. Example

We can overwrite a method or property added using the prototype method, as demonstrated in the below example.

```
// Function constructor
function Company(CompanyName, CompanyLocation, CompanyDomain)
{
  this.CompanyName = CompanyName;
  this.CompanyLocation = CompanyLocation;
  this.CompanyDomain = CompanyDomain;
}

// Adding a method to the constructor's prototype
Company.prototype.printCompanyDetails = function () {
  console.log(
    `${this.CompanyName} is located in
${this.CompanyLocation} and it is an ${this.CompanyDomain}
company`
  );
};

// Creating instances of Company
const company1 = new Company("PW Skills", "Banglore",
"Edtech");
company1.printCompanyDetails(); // PW Skills is located in
Banglore and it is an Edtech company

// Modifying the method using prototype
Company.prototype.printCompanyDetails = function () {
  console.log("Modified method");
};

const company2 = new Company("iNeuron", "Banglore",
"Edtech");
company1.printCompanyDetails(); // Output: Modified method
company2.printCompanyDetails(); // Output: Modified method
```

In the above example, we can see that the first log and the last two logs are different as we had modified the method after creating the company1 instance.
Even the company1 method is overwritten as the reference has been changed for it.

Polymorphism and prototype chains

Polymorphism is a fundamental concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. In JavaScript, which is a prototype-based language, polymorphism is achieved through inheritance and method overriding.

In JavaScript, we can have multiple objects with different prototypes and still call the same method on them. When a method is called on an object, JavaScript searches for that method in the object's prototype chain, going up the chain until it finds the method or reaches the top-level object. This thing we had seen in the above examples shows how it is checking for the methods and properties till it finds that else will return undefined.

Prototype chaining is a mechanism in JavaScript that allows objects to inherit properties and methods from their prototype objects. When you access a property or method on an object, JavaScript first checks if that property or method exists on the object itself. If not, it looks up the prototype chain to find the property or method in its prototype object, and so on, until it reaches the top-level object.

Note: We will discuss more about the polymorphism and inheritance in the OOJS classes.