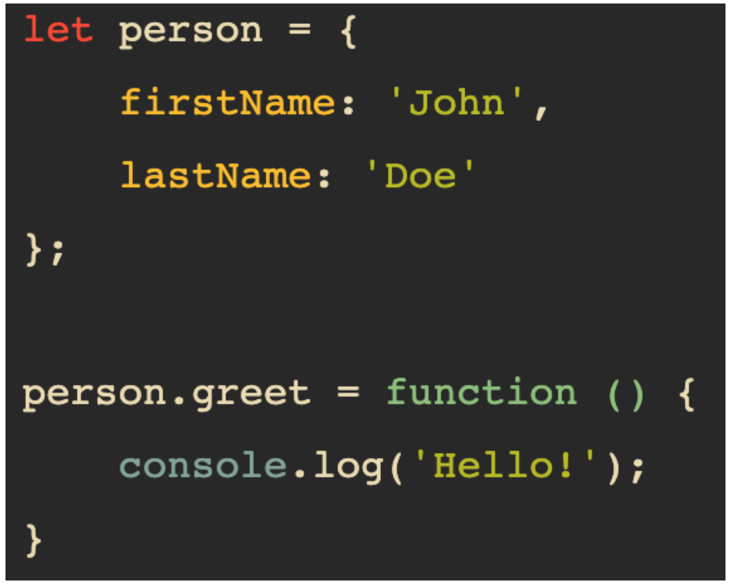
JavaScript Fundamentals

Instructor: Bui Binh Giang
buibinhgiang@vanlanguni.vn

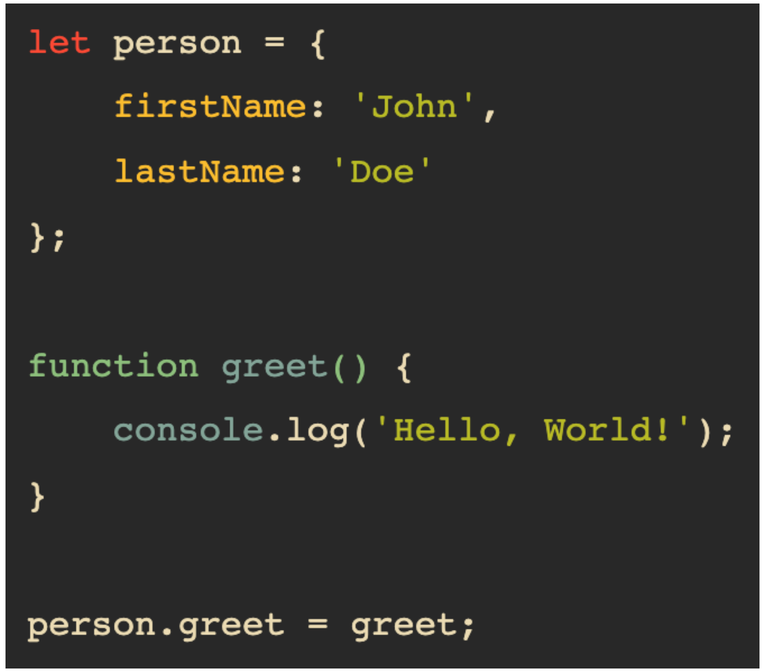
OBJECTS

JavaScript object methods

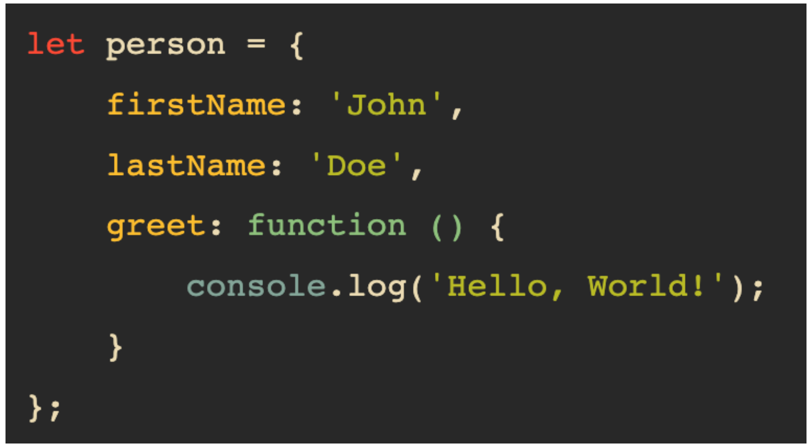
An **object** is a collection of key/value pairs or **properties**. When the value is a **function**, the property becomes a **method.**



```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};

person.greet = function () {
    console.log('Hello!');
}
```



```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};

function greet() {
    console.log('Hello, World!');
}

person.greet = greet;
```

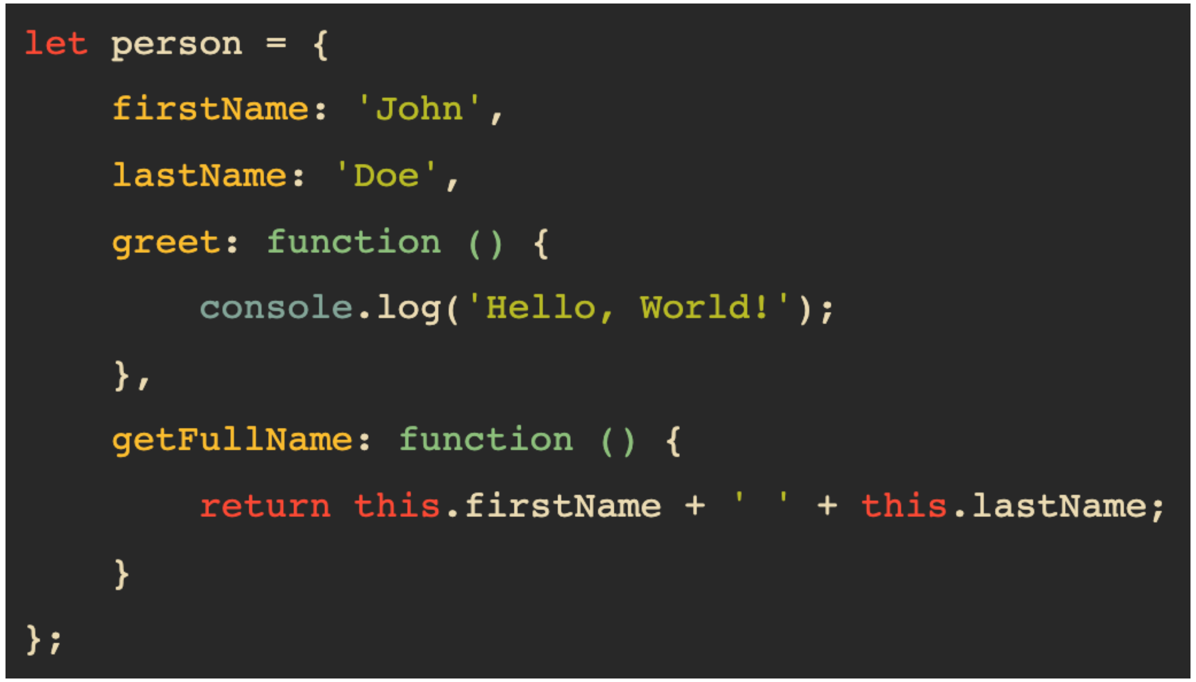


```
let person = {
    firstName: 'John',
    lastName: 'Doe',
    greet: function () {
        console.log('Hello, World!');
    }
};
```

JavaScript object methods - The ‘this’ value

Typically, methods need to access other properties of the object.

Inside a **method**, the **this** value references the **object** that invokes the **method**. Therefore, you can access an object’s property using the **this** value

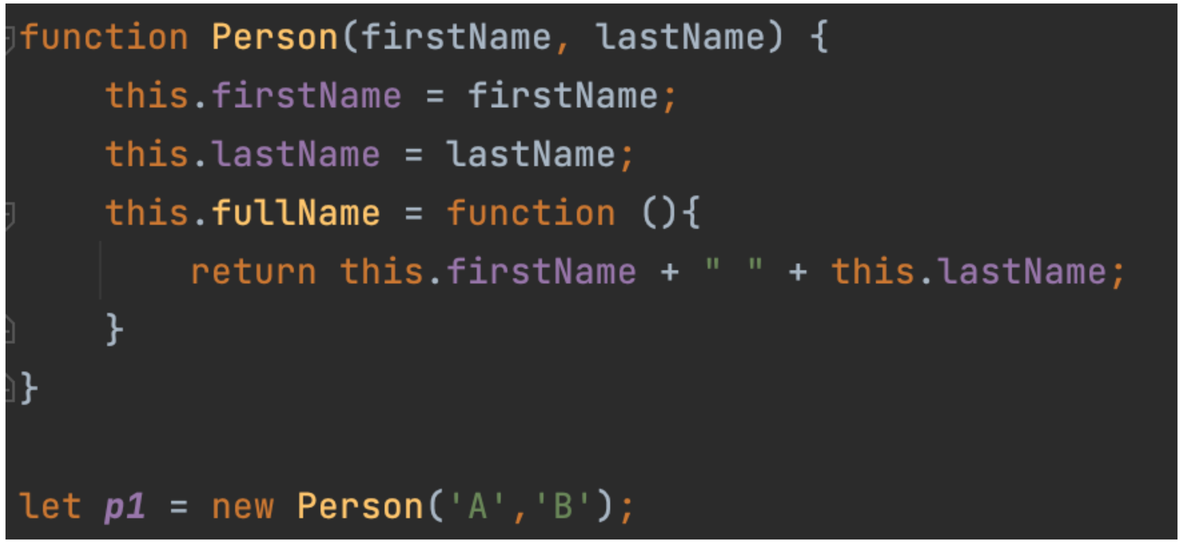


```
let person = {
    firstName: 'John',
    lastName: 'Doe',
    greet: function () {
        console.log('Hello, World!');
    },
    getFullName: function () {
        return this.firstName + ' ' + this.lastName;
    }
};
```

JavaScript Constructor Function

Constructor function help us to create similar objects. Technically, a **constructor function** is a regular function with the following convention:

- The name of a **constructor function** starts with a capital letter. Ex: Person, Document
- A **constructor function** should be called only with the **new** operator. The **new** operator creates a new empty object and assign it to the this variable



```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.fullName = function (){
        return this.firstName + " " + this.lastName;
    }
}

let p1 = new Person('A','B');
```

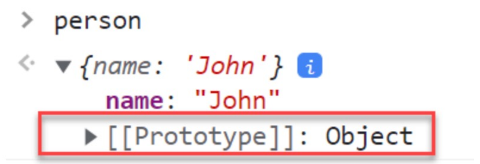
PROTOTYPE

JavaScript prototype

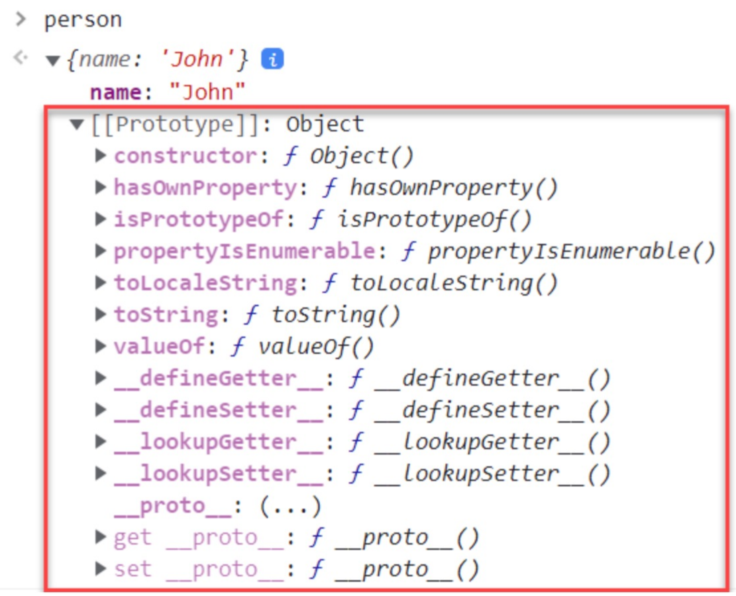
In JavaScript, every **object** has its own property called **prototype. Objects** can inherit features from one another via **prototypes**.

Because a **prototype** itself is also an **object** → the **prototype** has its own **prototype** → This creates a something called **prototype chain**. The **prototype chain** ends when a **prototype** has **null** for its own prototype.

```javascript
let person = {'name' : 'John'}
```



```
> person
< ▼{name: 'John'} i
      name: "John"
    ▶ [[Prototype]]: Object
```

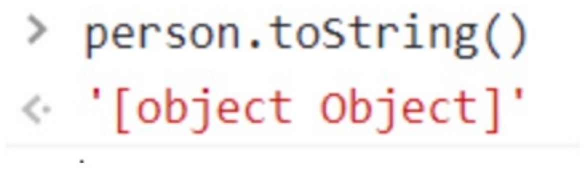


```
> person
< ▼{name: 'John'} i
      name: "John"
    ▼ [[Prototype]]: Object
      ▶ constructor: ƒ Object()
      ▶ hasOwnProperty: ƒ hasOwnProperty()
      ▶ isPrototypeOf: ƒ isPrototypeOf()
      ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶ toLocaleString: ƒ toLocaleString()
      ▶ toString: ƒ toString()
      ▶ valueOf: ƒ valueOf()
      ▶ __defineGetter__: ƒ __defineGetter__()
      ▶ __defineSetter__: ƒ __defineSetter__()
      ▶ __lookupGetter__: ƒ __lookupGetter__()
      ▶ __lookupSetter__: ƒ __lookupSetter__()
        __proto__: (...)
      ▶ get __proto__: ƒ __proto__()
      ▶ set __proto__: ƒ __proto__()
```

JavaScript prototype

When you access a property of an object:

- If the object has that property, it'll return the property value.
- If the object hasn't that property, the JavaScript engine will search in the **prototype** of the object → If the JavaScript engine cannot find the property in the object's **prototype**, it'll search in the **prototype's prototype** until either it finds the property or reaches the end of the prototype chain



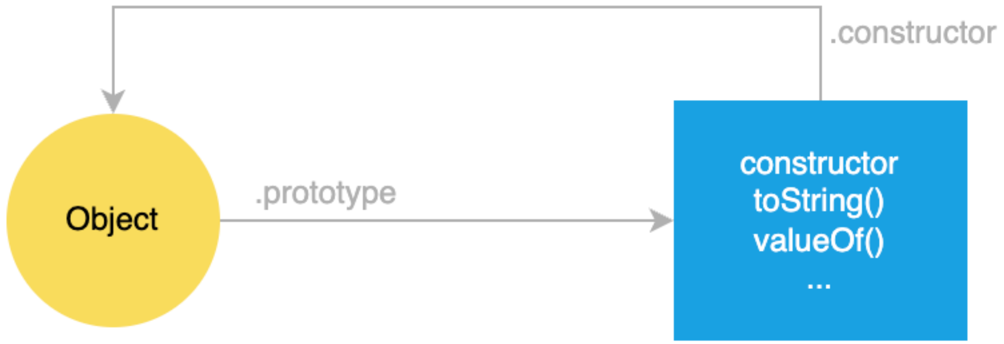
```
> person.toString()
<· '[object Object]'
```

JavaScript prototype illustration

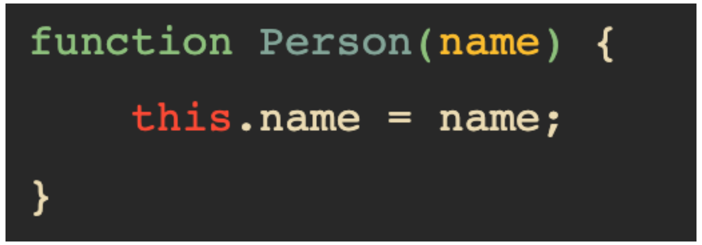
JavaScript has the built-in **Object()** function → **Object()** function has its own property called **prototype**

The **Object.prototype** is a object, it has some useful properties and methods such as **toString**() and **valueOf**()

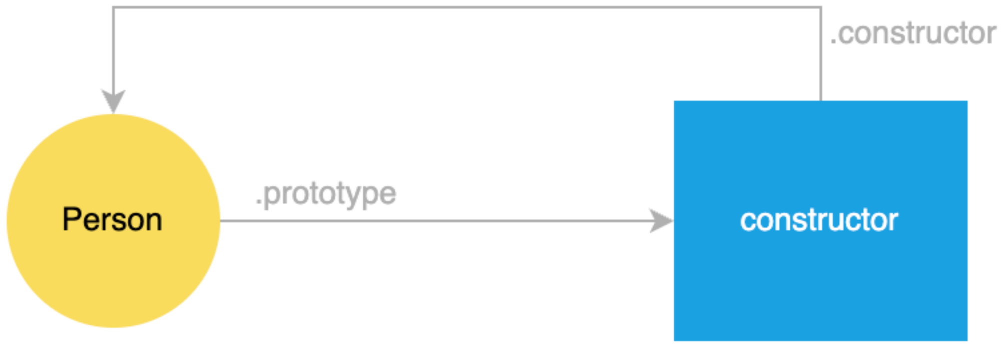
The **Object.prototype** also has an important property called **constructor** that references the **Object()** function



JavaScript prototype illustration

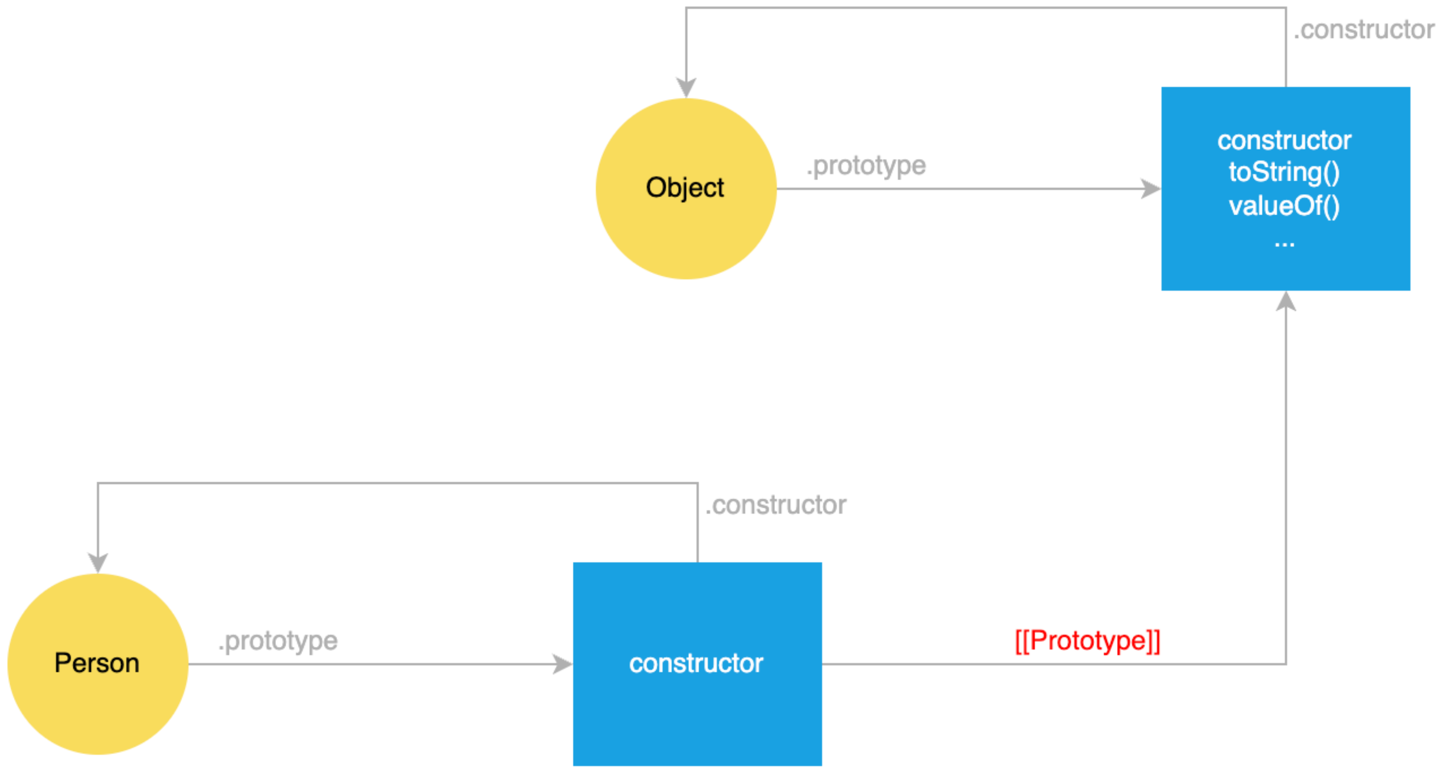


```
function Person(name) {
    this.name = name;
}
```



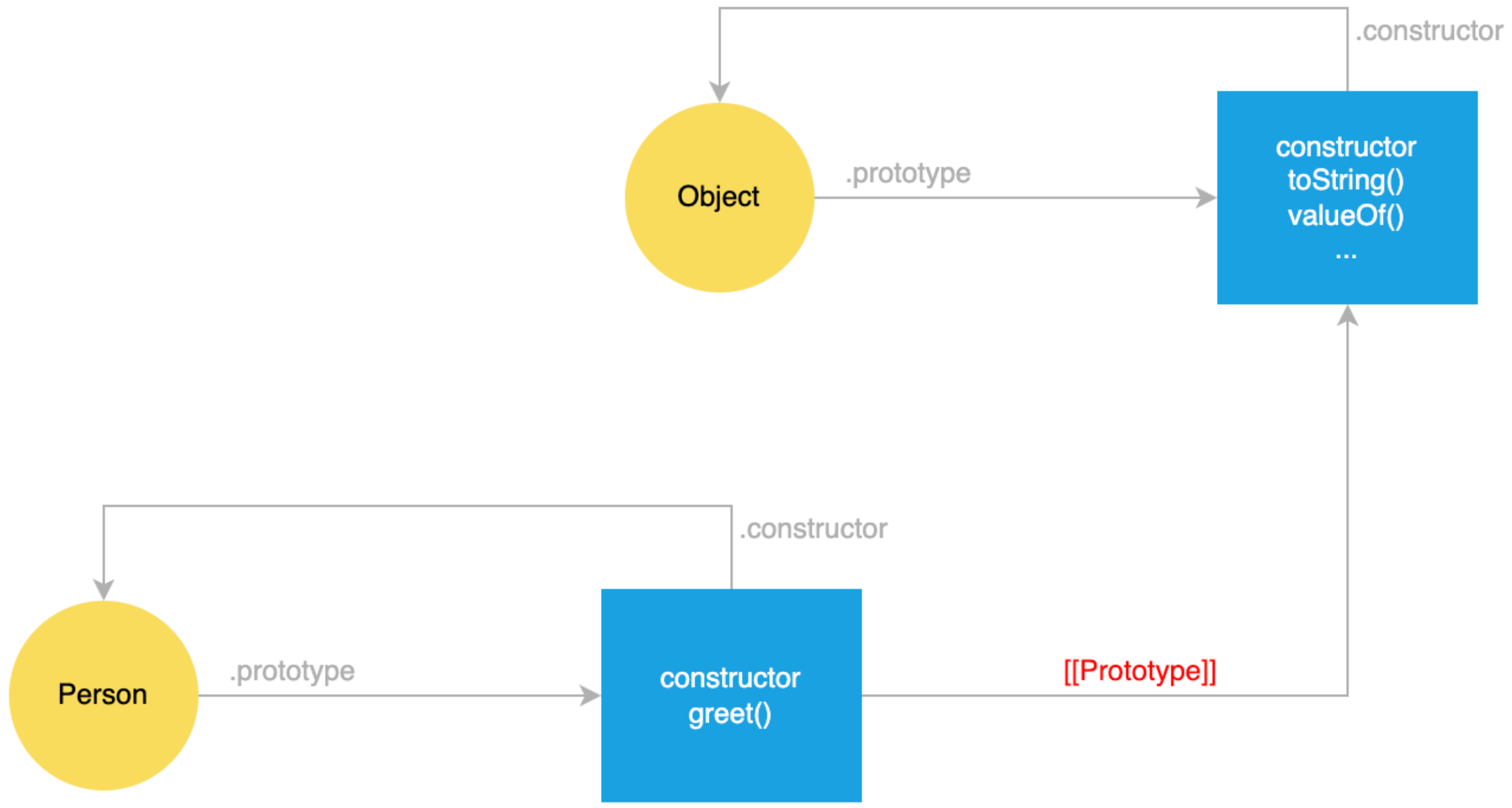
JavaScript prototype illustration

JavaScript links the **Person.prototype** object to the **Object.prototype** object via the **[[Prototype]]**, which is known as a ***prototype linkage***



Defining methods in the JavaScript prototype object

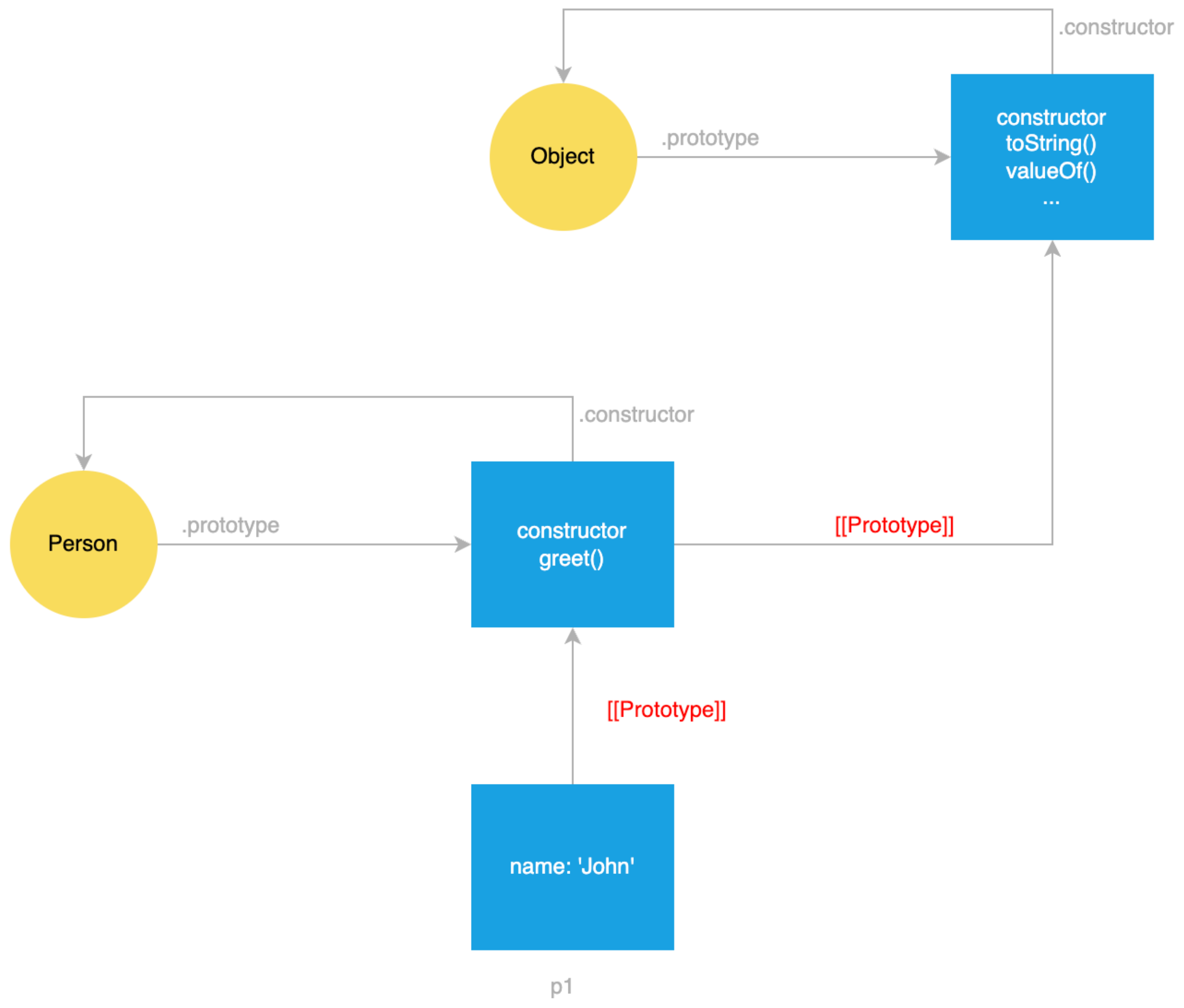
```
Person.prototype.greet = function() {
    return "Hi, I'm " + this.name + "!";
}
```



Defining methods in the JavaScript prototype object

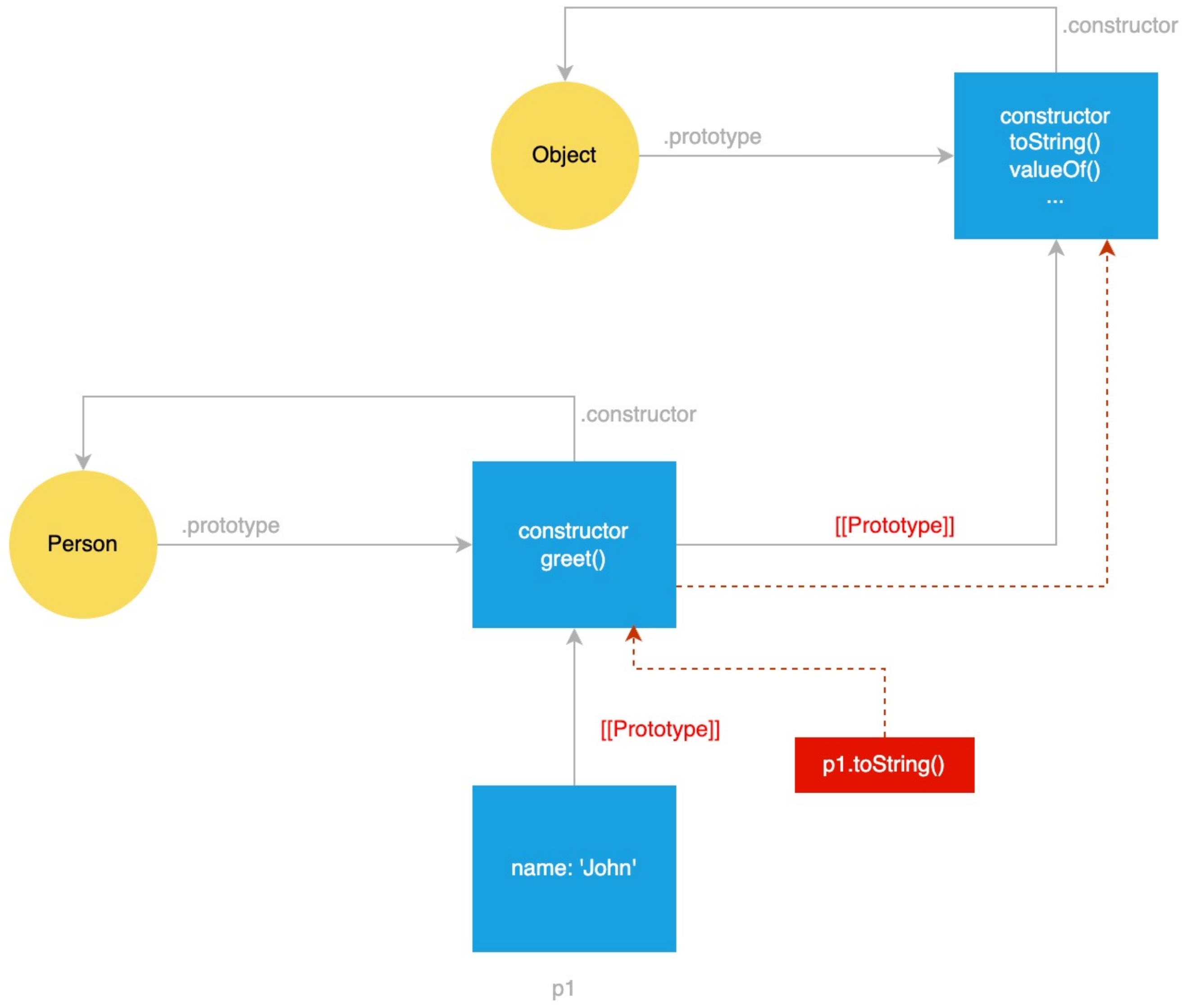
```
let p1 = new Person('John');
```

```
let greeting = p1.greet();
console.log(greeting);
```



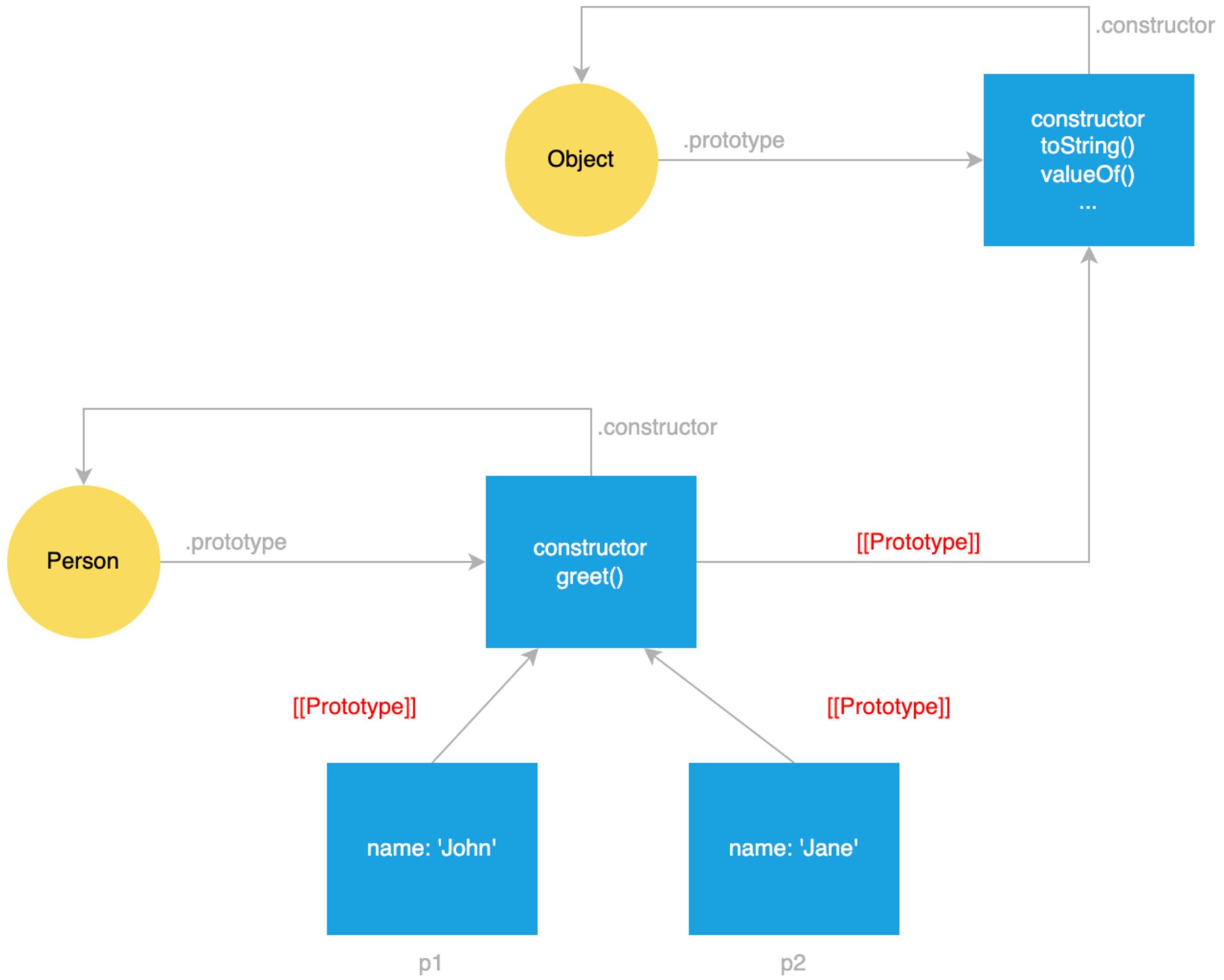
Defining methods in the JavaScript prototype object

```
let s = p1.toString();
console.log(s);
```



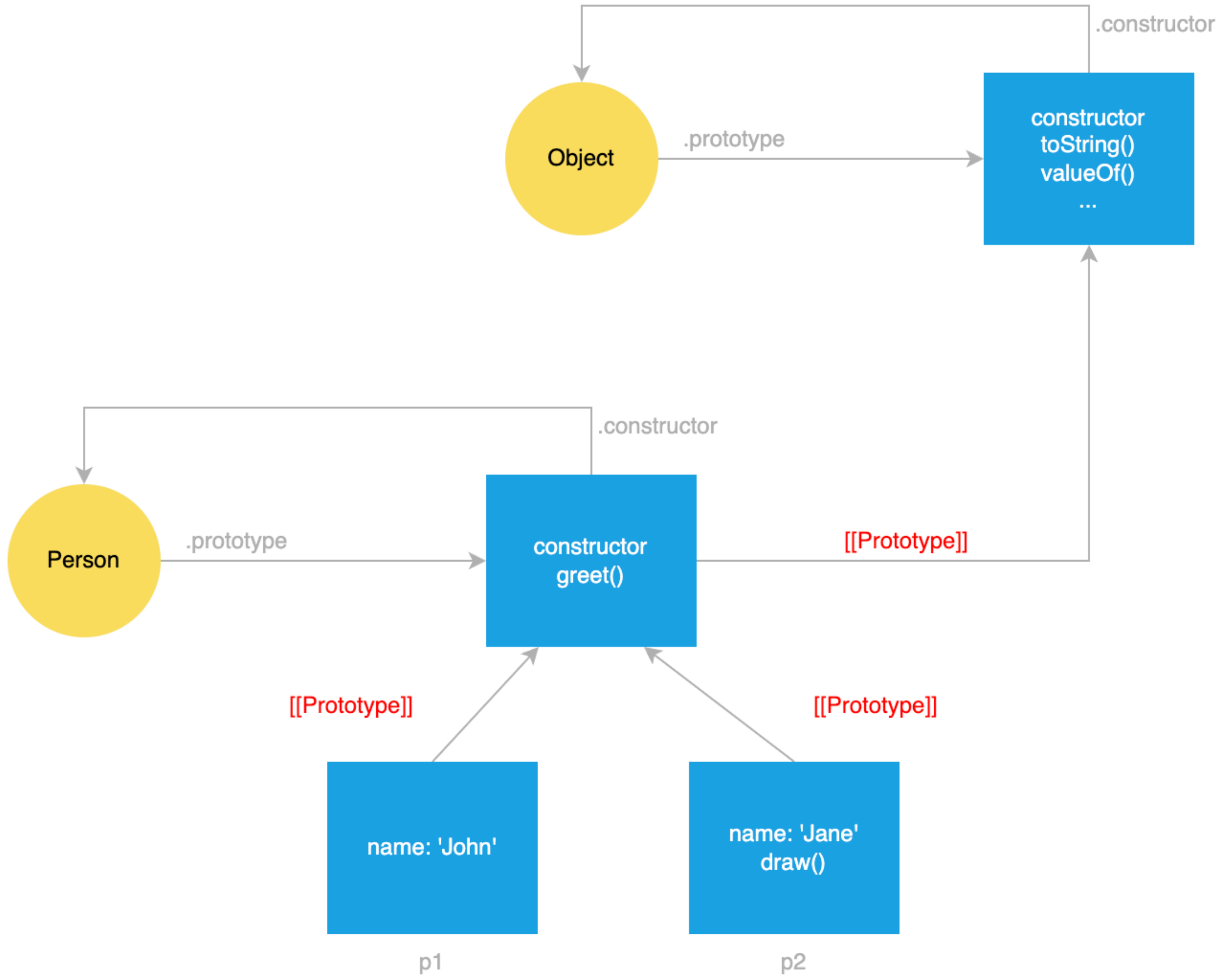
Defining methods in the JavaScript prototype object

```
let p2 = new Person('Jane');
```

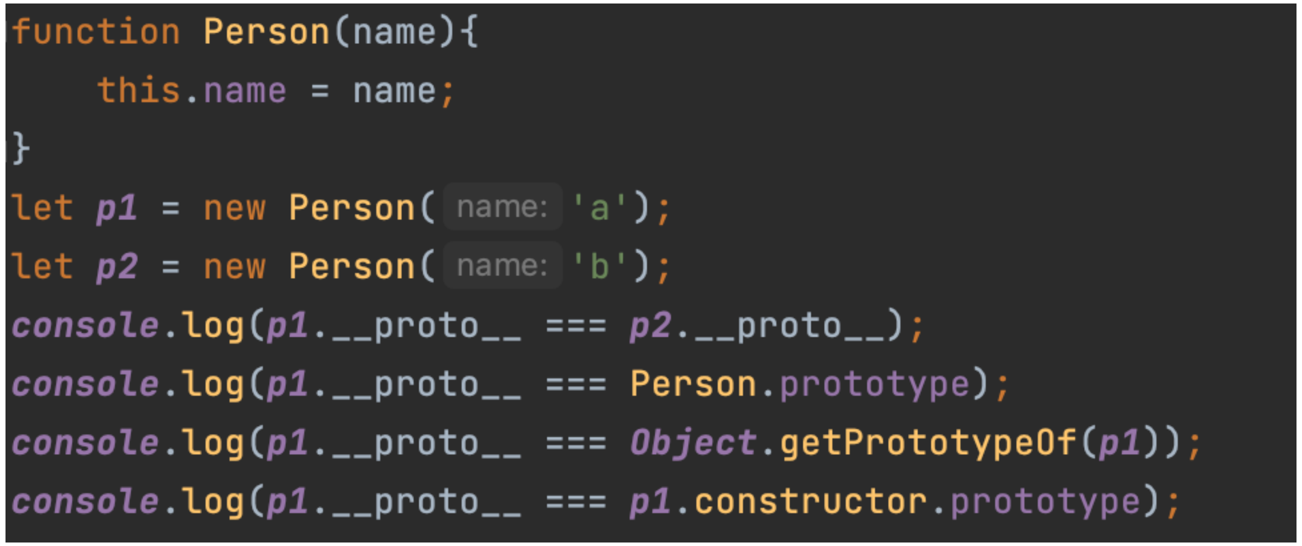


Defining methods in an individual object

```
p2.draw = function () {
    return "I can draw.";
};
```



Getting prototype linkage



```
function Person(name){
    this.name = name;
}
let p1 = new Person( name: 'a');
let p2 = new Person( name: 'b');
console.log(p1.__proto__ === p2.__proto__);
console.log(p1.__proto__ === Person.prototype);
console.log(p1.__proto__ === Object.getPrototypeOf(p1));
console.log(p1.__proto__ === p1.constructor.prototype);
```

Exercise - 301

Requirement:

- Write a Car constructor that initializes “**model**” and “**kmPerLitre**” from arguments.
- All instances built with Car:
  - Should initialize with an “**tank**” at 0
  - Should initialize with an “**odometer**” at 0
- Give cars the ability to get fueled with a “**.fill(litres)**” method. Add the fuel to “**tank**”.
- Give cars ability to “**.drive(distance)**”. The distance driven:
  - Should cause the “**odometer**” to go up.
  - Should cause the the “**tank**” to go down taking “**kmPerLitre**” into account.
  - A car which runs out of fuel while driving can't drive any more distance and the “**.drive**” method should return a string “Vehicle ran out of gas after driving **x** km. Vehicle has gone **y** km and **z** litre(s) left”.

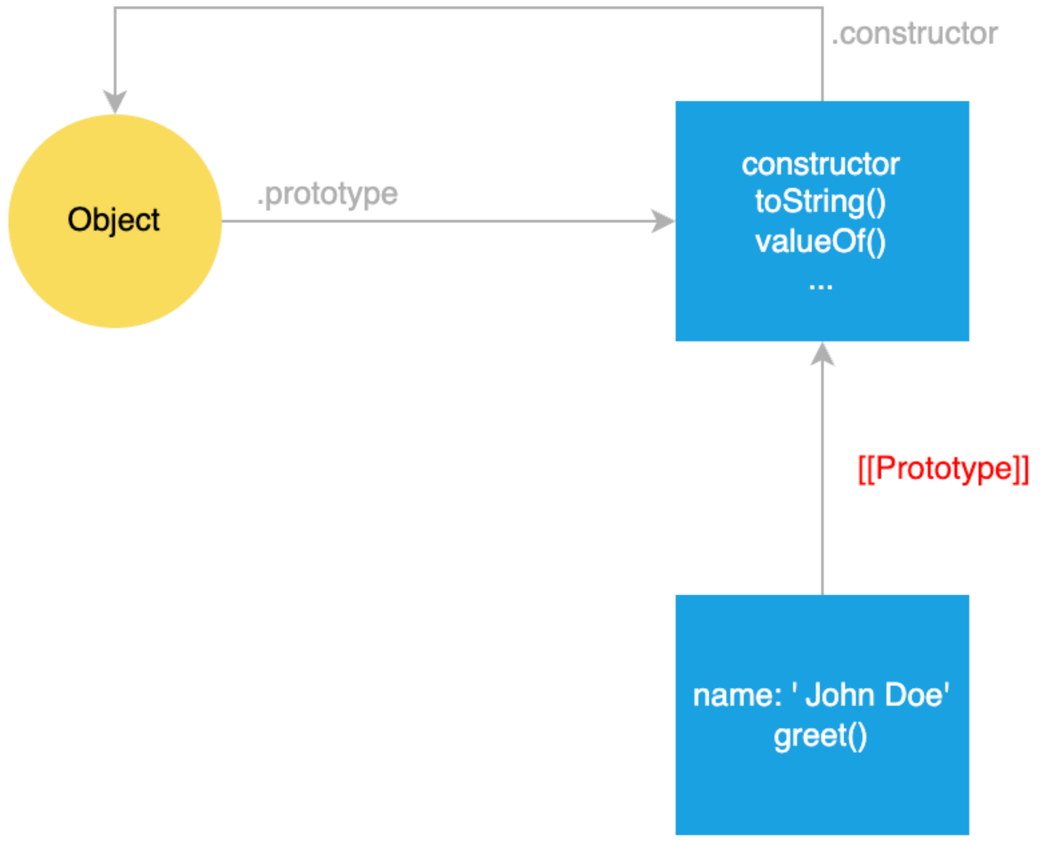
JavaScript Prototypal Inheritance

- Inheritance allows an object to use the properties and methods of another object without duplicating the code.
- Object-oriented programming languages such as Java or C++ use **classical inheritance**.
- JavaScript uses **prototypal inheritance**. In **prototypal inheritance**, an object "inherits" properties from another object via the **prototype linkage.**

JavaScript Prototypal Inheritance

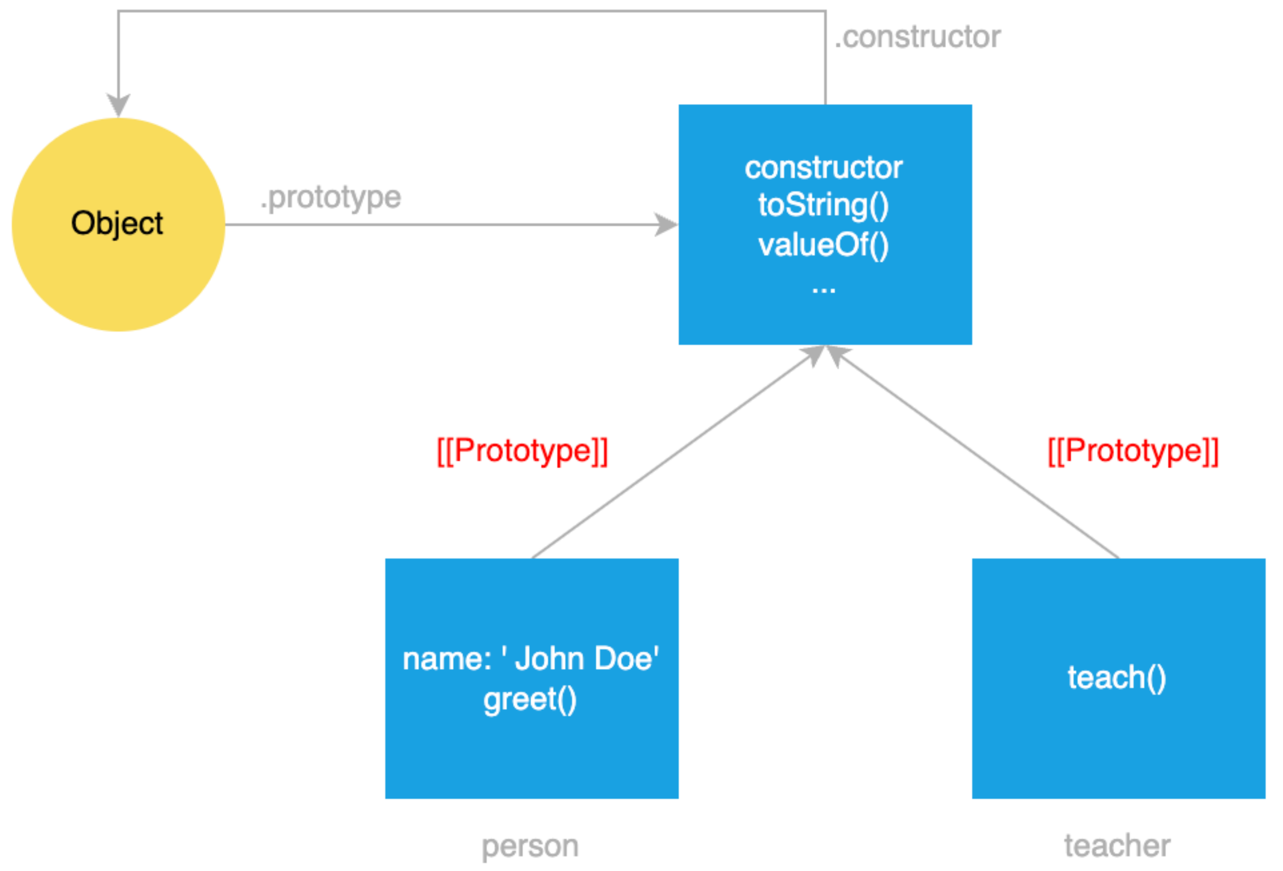
```
let person = {
    name: "John Doe",
    greet: function () {
        return "Hi, I'm " + this.name;
    }
};
```

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



JavaScript Prototypal Inheritance

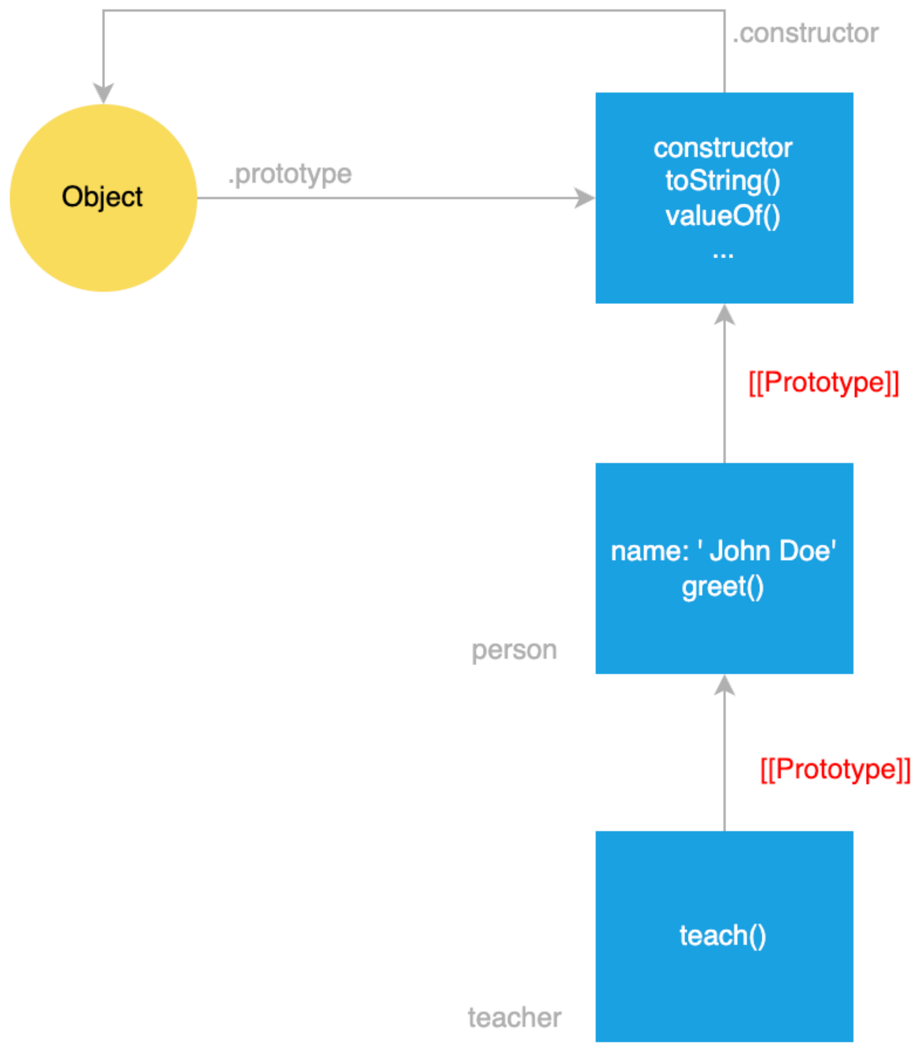
```
let teacher = {
    teach: function (subject) {
        return "I can teach " + subject;
    }
};
```



JavaScript Prototypal Inheritance

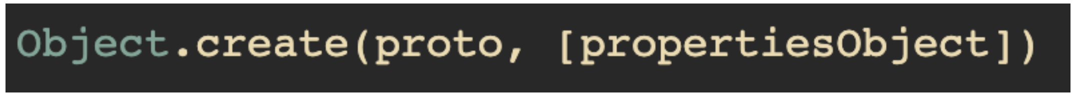
If you want the teacher object to access **all methods and properties** of the person object, you can set the **prototype** of teacher object to the person object

```
teacher.__proto__ = person;
```



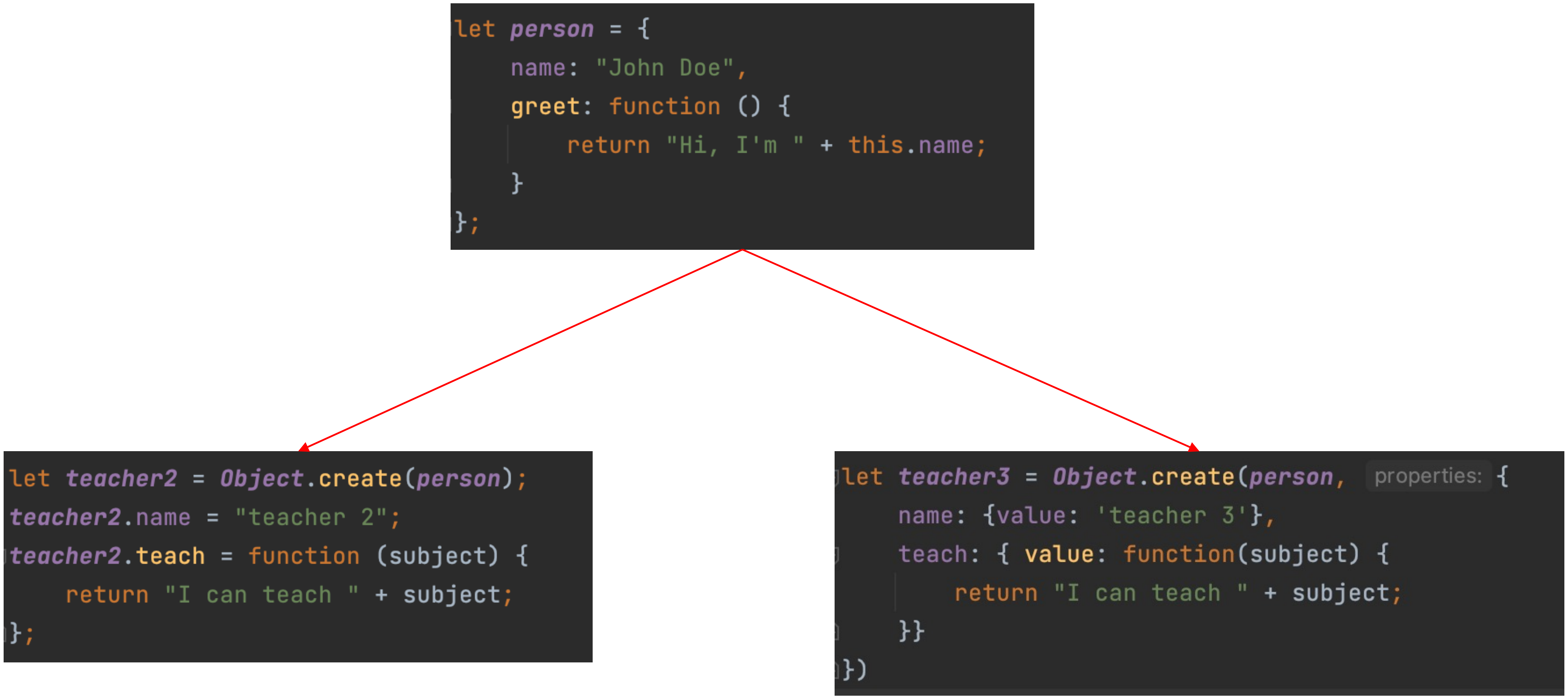
JavaScript Prototypal Inheritance in ES5

- **ES5** provided a standard way to work with **prototypal inheritance** by using the **Object.create()** method. The **Object.create()** method accepts two arguments:
- The first argument (**proto**) is an object used as the prototype for the new object.
- The second argument (**propertiesObject**), if provided, is an optional object that defines additional properties for the new object.



```
Object.create(proto, [propertiesObject])
```

JavaScript Prototypal Inheritance in ES5



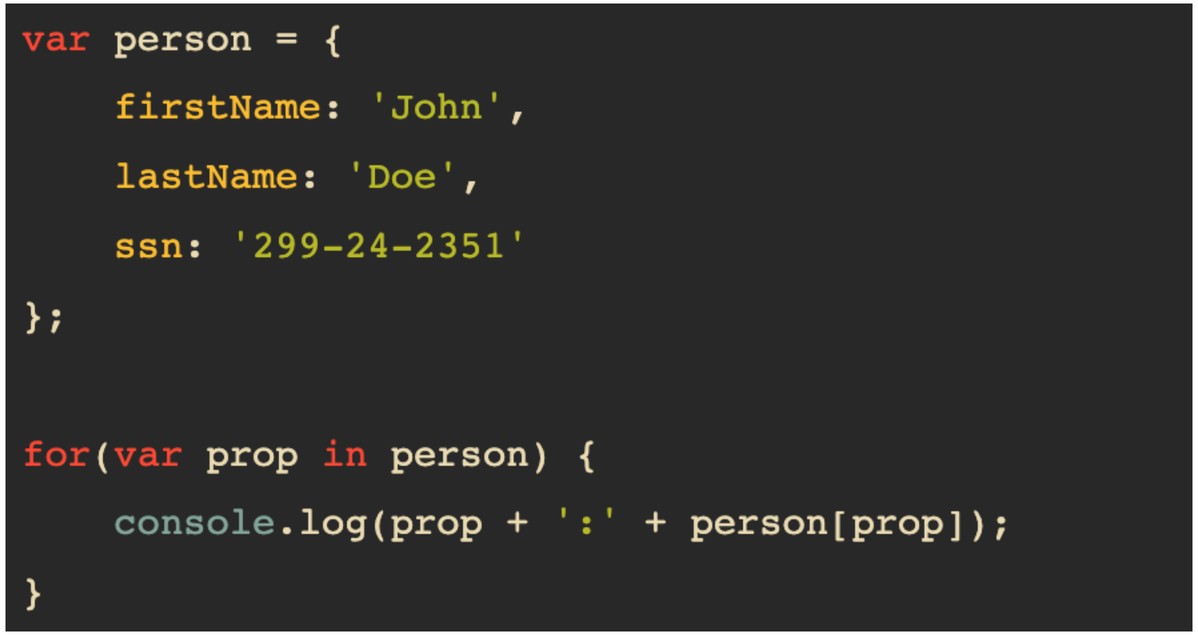
```
let person = {
    name: "John Doe",
    greet: function () {
        return "Hi, I'm " + this.name;
    }
};
```

```
let teacher2 = Object.create(person);
teacher2.name = "teacher 2";
teacher2.teach = function (subject) {
    return "I can teach " + subject;
};
```

```
let teacher3 = Object.create(person,  properties: {
    name: {value: 'teacher 3'},
    teach: { value: function(subject) {
        return "I can teach " + subject;
    }}
})
```

JavaScript for...in Loop

The **for...in** allows you to access each property and value of an object without knowing the specific name of the property.



```
var person = {
    firstName: 'John',
    lastName: 'Doe',
    ssn: '299-24-2351'
};

for(var prop in person) {
    console.log(prop + ':' + person[prop]);
}
```

JavaScript for...in Loop & Inheritance

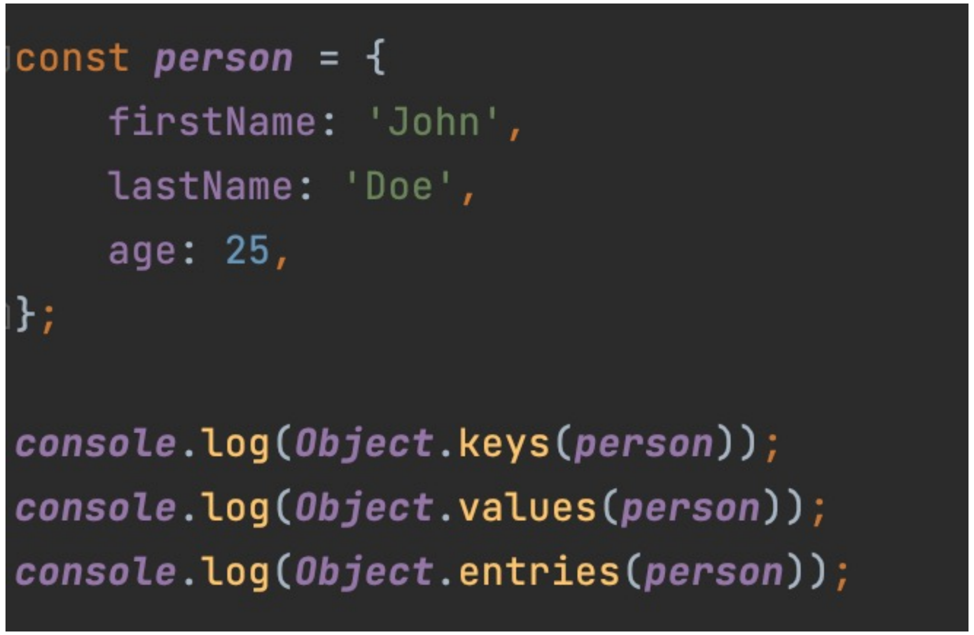
When you loop over the properties of an object that inherits from another object, the **for...in** statement goes up in the prototype chain and enumerates over inherited properties.

```
var decoration = {
    color: 'red'
};

var circle = Object.create(decoration);
circle.radius = 10;


for(const prop in circle) {
    console.log(prop);
}
```

```
radius
color
```

JavaScript Object.keys(), Object.values() , Object.entries()

- The **Object.keys()** accepts an object and returns its **own** property's names as an array.
- The **Object.values()** accepts an object and returns its **own** property's values as an array.
- The **Object.values()** accepts an object and returns its **own** string-keyed property **[key, value]** pairs of the object.



```
const person = {
    firstName: 'John',
    lastName: 'Doe',
    age: 25,
};

console.log(Object.keys(person));
console.log(Object.values(person));
console.log(Object.entries(person));
```

Exercise - 302

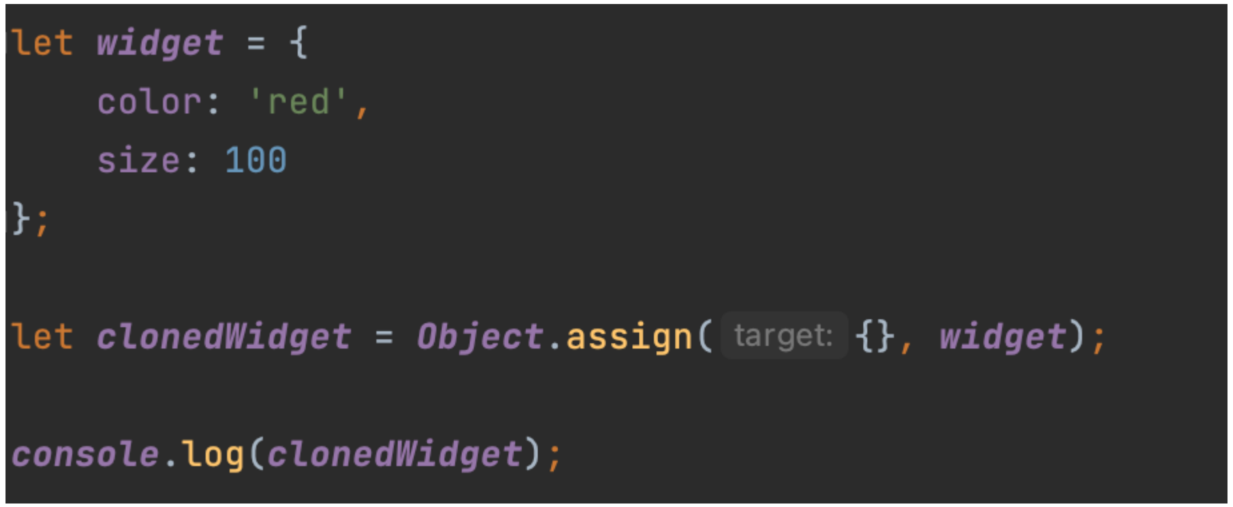
Requirement:

- Create an object named “**shape**” that has a property “**type**” and “**getType**” method.
- Write a **Triangle** constructor function whose prototype is **shape**. Objects created with **Triangle**() should have three own properties: **a**, **b**, and **c**, representing the lengths of the sides of a triangle.
- Add a new method to the prototype called ”.**getPerimeter**()”.
- Loop over an instance of Triangle showing only own properties and methods (none of the prototype's)

JavaScript Object.assign()

The **Object.assign()** copies all enumerable and own properties from the source objects to the target object. It returns the target object

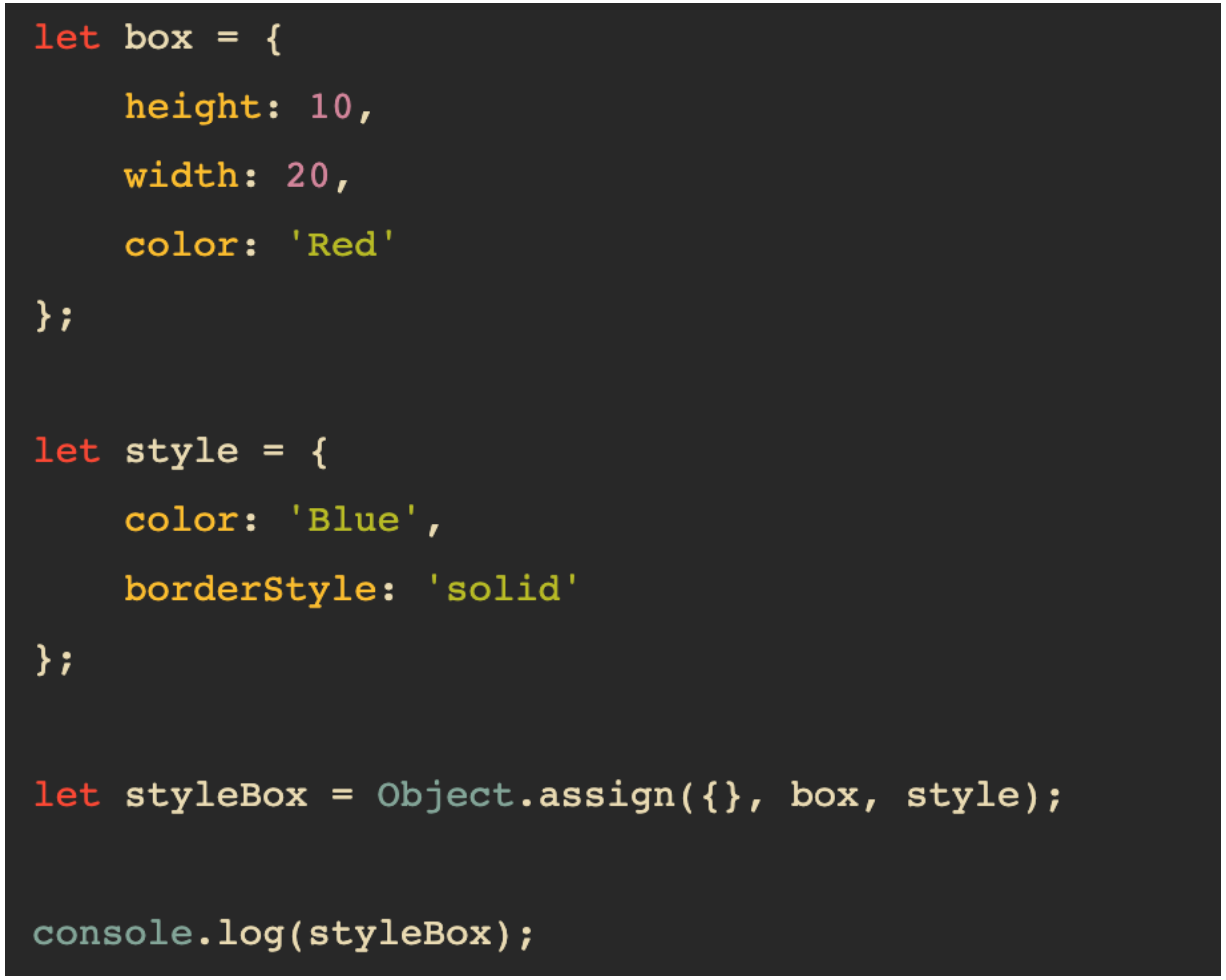
Using Object.assign() to clone an object



```
let widget = {
    color: 'red',
    size: 100
};

let clonedWidget = Object.assign( target: {}, widget);

console.log(clonedWidget);
```

JavaScript Object.assign()

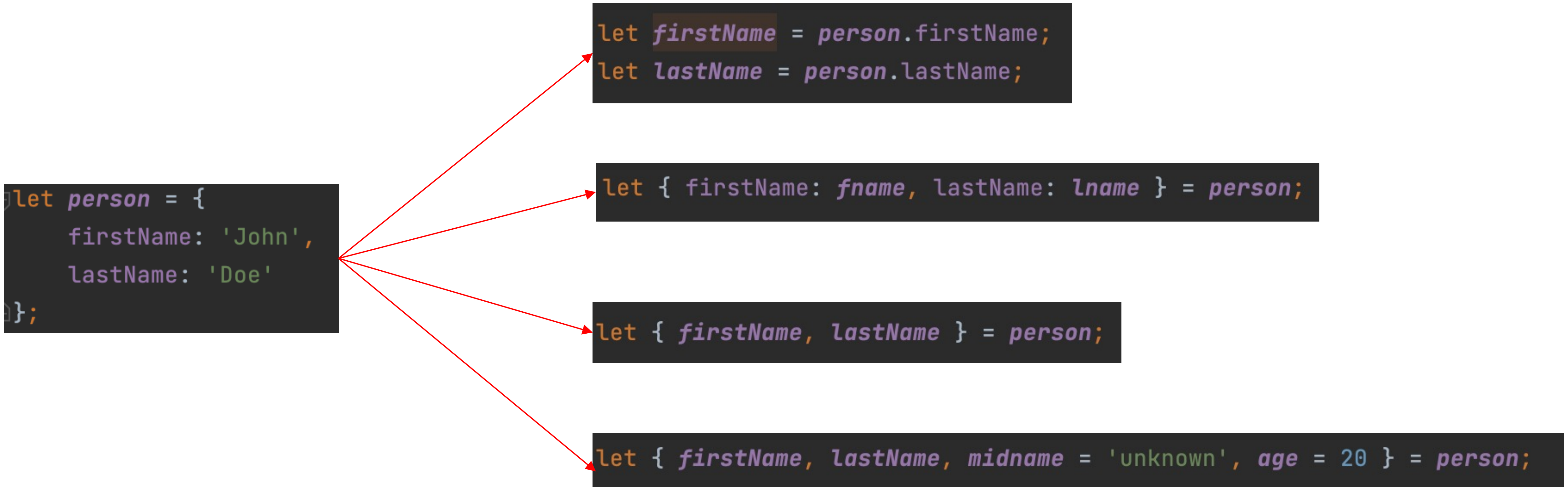
Using Object.assign() to merge objects



```
let box = {
    height: 10,
    width: 20,
    color: 'Red'
};

let style = {
    color: 'Blue',
    borderStyle: 'solid'
};

let styleBox = Object.assign({}, box, style);

console.log(styleBox);
```

JavaScript Object Destructuring

Object destructuring help us to assign properties of an object to individual variables



```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};
```

```
let firstName = person.firstName;
let lastName = person.lastName;
```

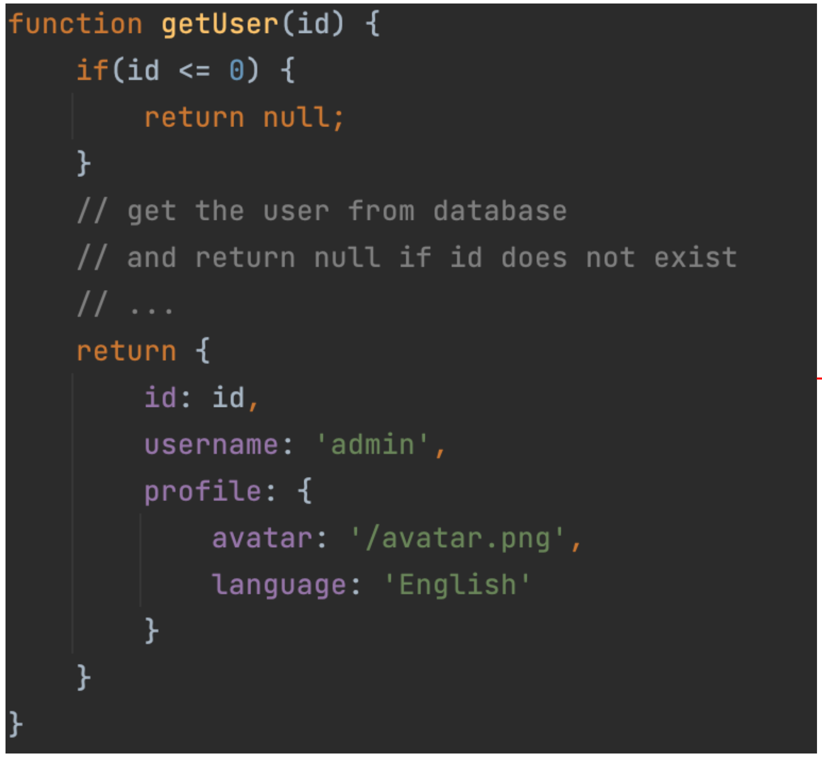
```
let { firstName: fname, lastName: lname } = person;
```

```
let { firstName, lastName } = person;
```

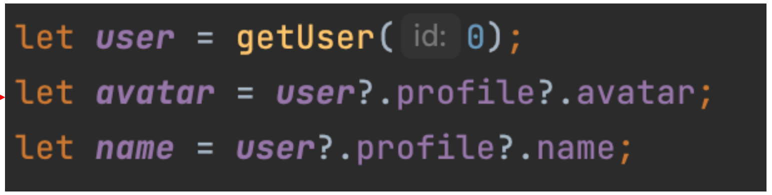
```
let { firstName, lastName, midname = 'unknown', age = 20 } = person;
```

JavaScript optional chaining operator

The **optional chaining operator (?.)** allows you to access the value of a property located deep within a chain of objects without explicitly checking if each reference in the chain is **null** or **undefined** → If one of the references in the chain is **null** or **undefined**, the **optional chaining operator (?.)** will short circuit and return **undefined**



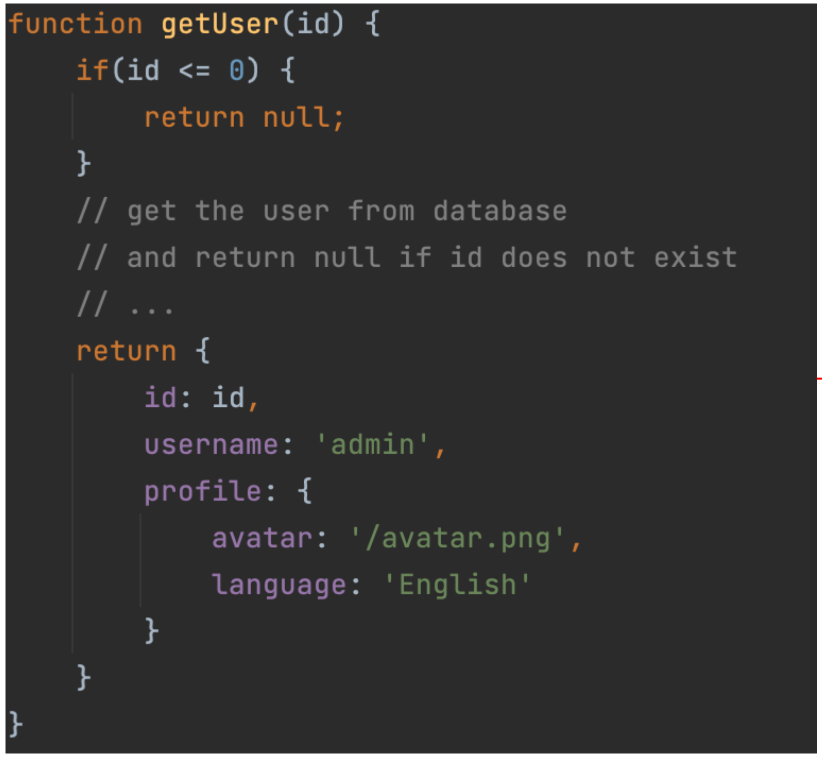
```
function getUser(id) {
    if(id <= 0) {
        return null;
    }
    // get the user from database
    // and return null if id does not exist
    // ...
    return {
        id: id,
        username: 'admin',
        profile: {
            avatar: '/avatar.png',
            language: 'English'
        }
    }
}
```



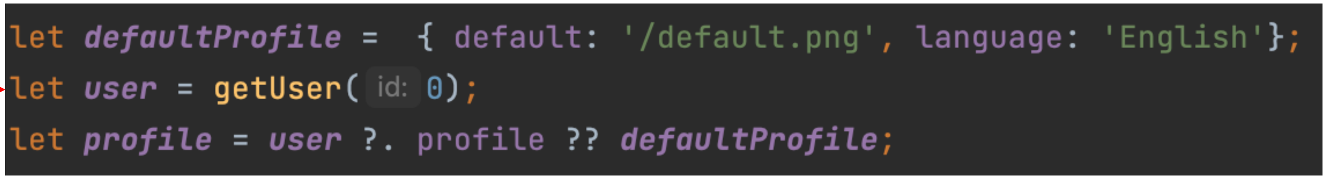
```
let user = getUser( id: 0);
let avatar = user?.profile?.avatar;
let name = user?.profile?.name;
```

JavaScript optional chaining operator in Combining with the nullish coalescing operator

The **optional chaining operator (?.)** allows you to access the value of a property located deep within a chain of objects without explicitly checking if each reference in the chain is **null** or **undefined** → If one of the references in the chain is **null** or **undefined**, the **optional chaining operator (?.)** will short circuit and return **undefined**

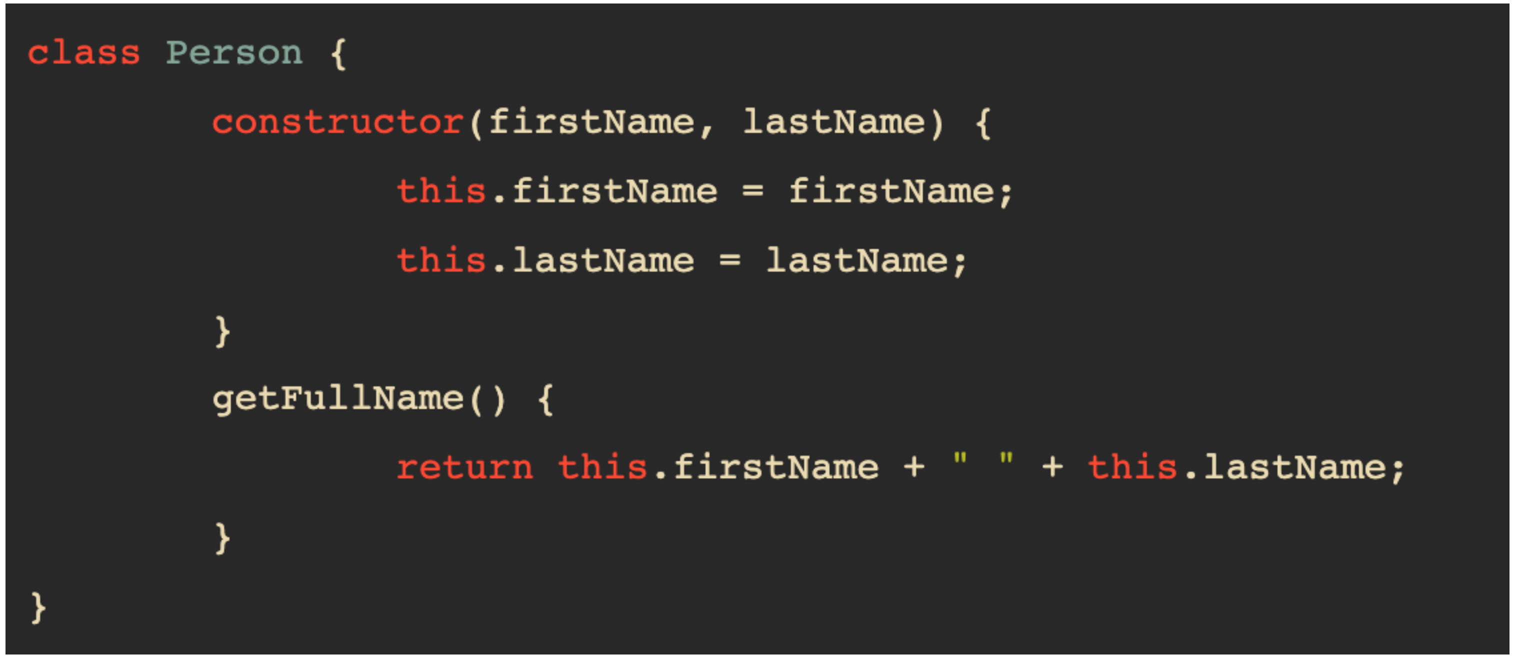


```
function getUser(id) {
    if(id <= 0) {
        return null;
    }
    // get the user from database
    // and return null if id does not exist
    // ...
    return {
        id: id,
        username: 'admin',
        profile: {
            avatar: '/avatar.png',
            language: 'English'
        }
    }
}
```



```
let defaultProfile =  { default: '/default.png', language: 'English'};
let user = getUser( id: 0);
let profile = user ?. profile ?? defaultProfile;
```

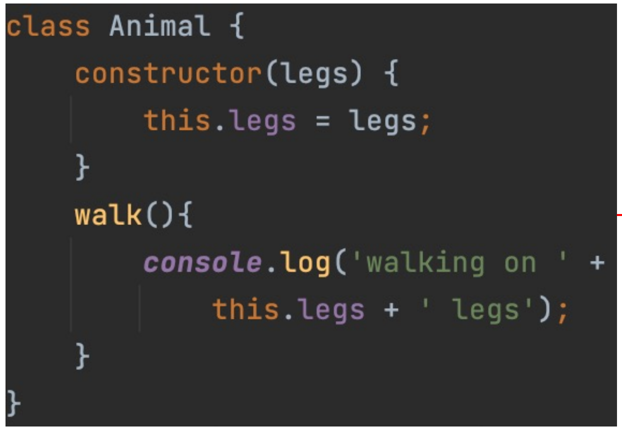
JavaScript ES6 Class



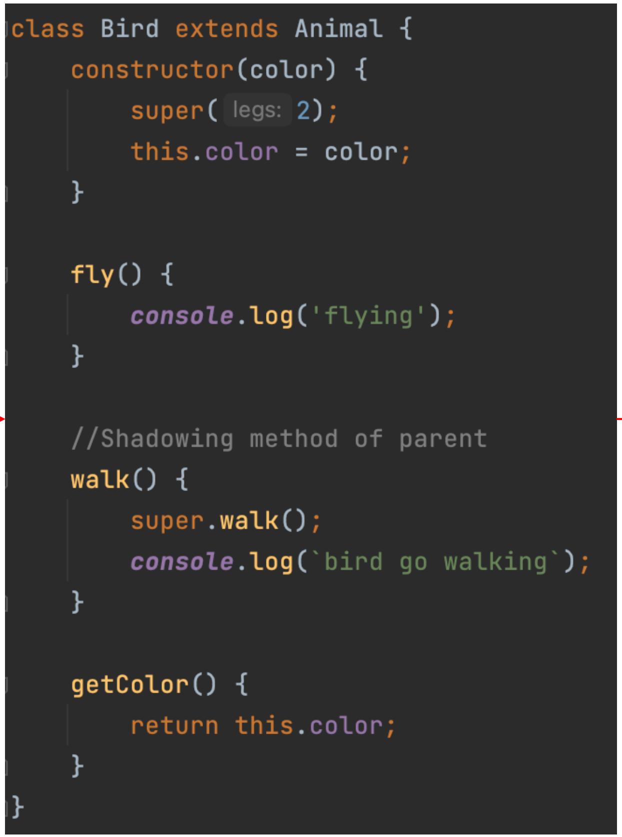
```
class Person {
        constructor(firstName, lastName) {
                this.firstName = firstName;
                this.lastName = lastName;
        }
        getFullName() {
                return this.firstName + " " + this.lastName;
        }
}
```

JavaScript Class Inheritance Using extends & super

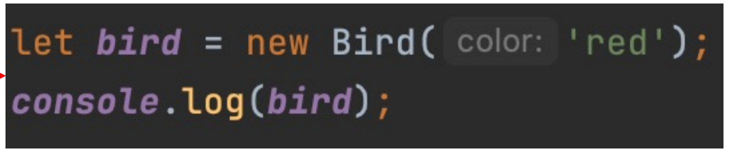
- Use the **extends** keyword to implement the **inheritance** in ES6. The class to be extended is called a **base class** or **parent class**. The class that extends the **base class** or **parent class** is called the **derived class** or **child class.**
- The **super(arguments)** must be called in the **child class's constructor** to invoke the **parent class's constructor**.



```
class Animal {
    constructor(legs) {
        this.legs = legs;
    }
    walk(){
        console.log('walking on ' +
            this.legs + ' legs');
    }
}
```



```
class Bird extends Animal {
    constructor(color) {
        super( legs: 2);
        this.color = color;
    }

    fly() {
        console.log('flying');
    }

    //Shadowing method of parent
    walk() {
        super.walk();
        console.log(`bird go walking`);
    }

    getColor() {
        return this.color;
    }
}
```



```
let bird = new Bird( color: 'red');
console.log(bird);
```

Exercise - 303

Requirement:

- Create a Person class that initializes “**firstName**”, “**lastName**” and “**age**” from arguments.
    - Give Person ability to “.**getFullname**()”
- Create Student class that extend from Person.
    - All instances of Student should initialize with an empty “**bag**” .
    - Give students ability to “.**putItem**(item)” into the bag. If there’re 3 items in the bag, the method should have no effect.
    - Give students ability to “.**getItem**(item)” from the bag.
    - Give students ability to empty the bag.
- Create Baby class that extend from Person.
    - All instances of Baby should initialize with “**favoriteToy**”.
    - Give students ability to “.**play**()” the favoriteToy.