Objects and object constructor

Tuesday, July 12, 2022 10:03 AM

≠ objects and constructors

⇒ Let's take an example first

→ Let's say we have a game in which many players can play simultaneously, now we have to store their name and age in an object

→ Here we have two approaches

1) For every player declare new object for name & age

example

```
const playerOneName = "tim"
const playerTwoName = "jenn"
const playerOneMarker = "X"
const playerTwoMarker = "O"
```

⇒ Here we have a problem, we need to create new object for name and age for every new user, but we cannot remember exact variable of every user

⇒ And we are repeating the same steps again and again.

2) In second approach we will create an object and we will store name and age in that object only.

```
const playerOne = {
  name: "tim",
  marker: "X"
}

const playerTwo = {
  name: "jenn",
  marker: "O"
}
```

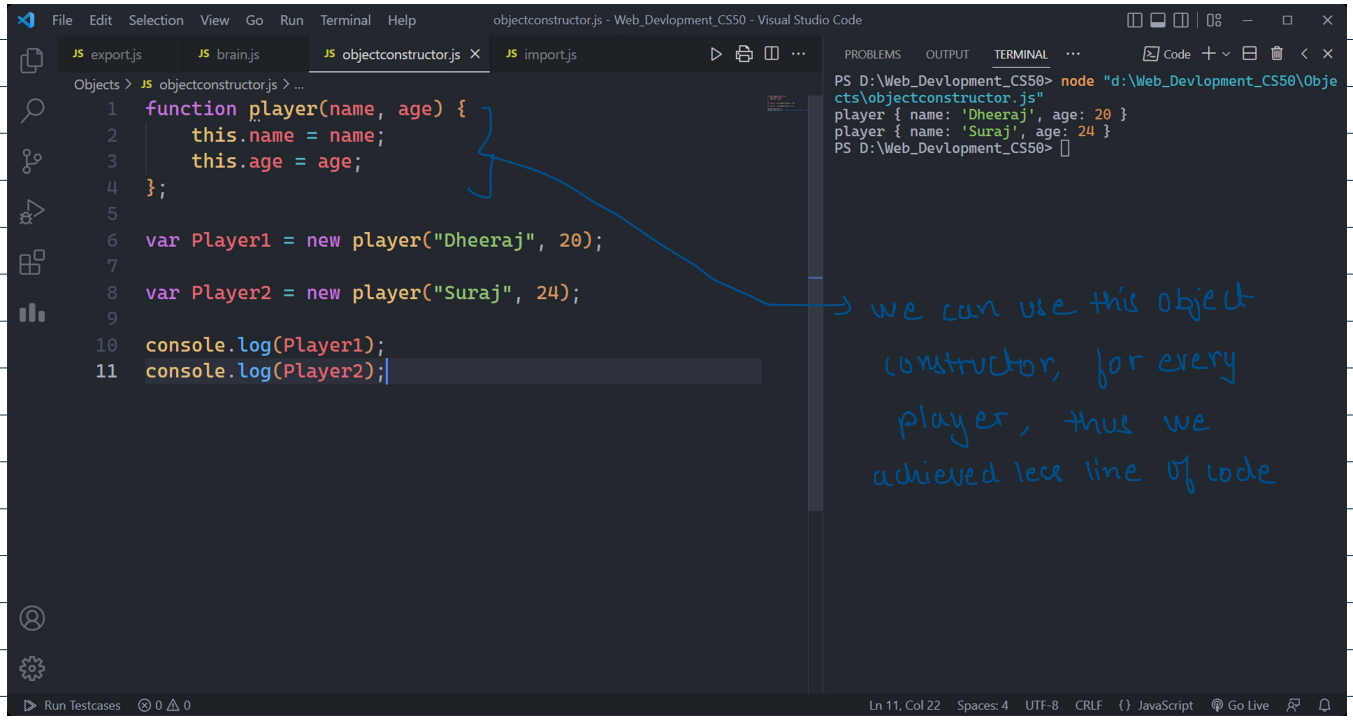
⇒ Here, again we need to repeat steps for every object.

3)

Object constructor: In JavaScript, there is a special constructor function known as **Object()** is used to create and initialize an object. The return value of the **Object()** constructor is assigned to a variable. The variable contains a reference to

the new object. We need an object constructor to create an object "type" that can be used multiple times without redefining the object every time.

From <https://www.geeksforgeeks.org/javascript-object-constructors/>



Exercise

Write a constructor for making "Book" objects. We will revisit this in the project at the end of this lesson. Your book objects should have the book's title, author, the number of pages, and whether or not you have read the book.

Put a function into the constructor that can report the book info like so:

theHobbit.info() // "The Hobbit by J.R.R. Tolkien, 295 pages, not read yet"

books.js - Web_Devlopment_CS50 - Visual Studio Code

xport.js | brain.js | objectconstructor.js | books.js | import.js

Objects > books.js > book > info

```
function book(title, author, pages, read) {
    this.title = title; // book title
    this.author = author; // author of the book
    this.pages = pages; // no of pages in book
    this.read = read; // whether book has already read or
    not

    this.info = function () {  // function to return all
    details
        var ans = `${this.title} by ${this.author}, ${this.
        pages} pages, `;
        if (read) { // if book is already read
            ans = ans + "already read";
        }
        else {
            ans = ans + "not read yet";
        }
        return ans;
    }
}

var thehobbit = new book("The Hobbit", "J.R.R. Tolkien",
295, false);

console.log(thehobbit.info());
```

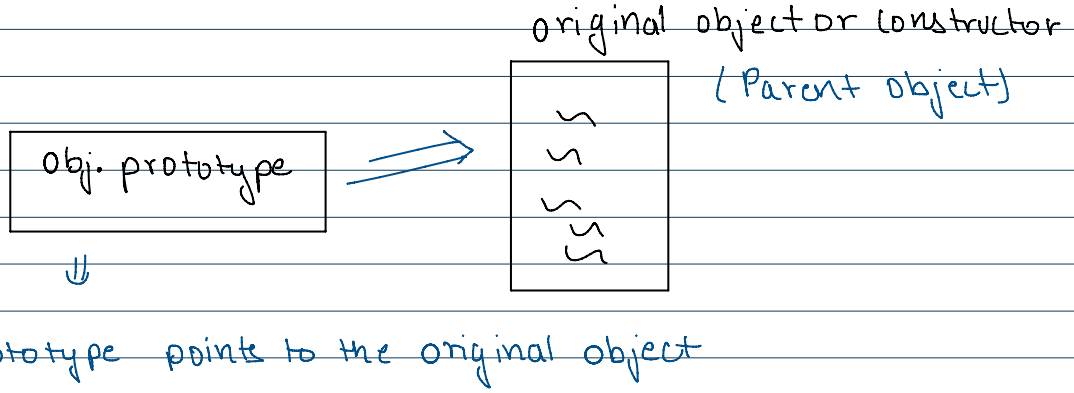
```
PS D:\Web_Devlopment_CS50> node "d:\Web_Devlopment_CS50\Objects\books.js"
The Hobbit by J.R.R. Tolkien, 295 pages, not read yet
PS D:\Web_Devlopment_CS50>
```

Run Testcases | Ln 16, Col 6 | Spaces: 4 | UTF-8 | CRLF | JavaScript | Go Live

Prototype

★ https://web.archive.org/web/20200513181548/https://javascriptissexy.com/javascript-prototype-in-plain-detailed-language/

⇒ Prototype :- It is the parent object from where the current object inherits all his property & "obj.prototype" is pointer to the parent object.



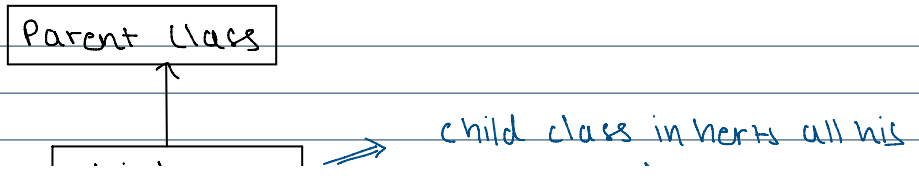
obj.prototype points to the original object

⇓

If we make any change through .prototype pointer it will be reflected to all instances of that object.

⇒ In Javascript we do not have classical inheritance.

i.e



child class ⟹ child class inherits all his properties from parent class

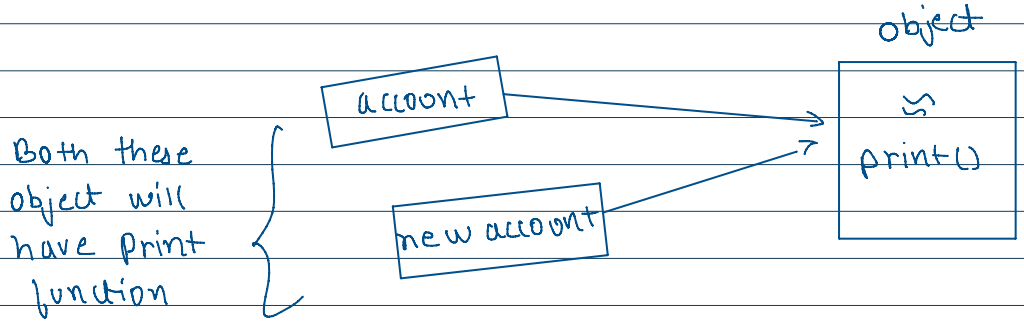
⇒ So, we have prototype in Javascript to use inheritance property of object.

```
var account = new Object();

Object.prototype.print = function () {
    console.log("This is an Object");
}

account.print();

console.log(account.constructor);
```

⟹ object is created

⇒ A print function is added, using prototype object

Now this print function will be associated with all instances of Object



⇒ We can also use Heirarchical inheritance in Java Script

Example

```
function Plant () {
this.country = "Mexico";
this.isOrganic = true;
}
// Add the showNameAndColor method to the
Plant prototype property
Plant.prototype.showNameAndColor =  function ()
{
console.log("I am a " + this.name + " and my color
is " + this.color);
}
// Add the amIOrganic method to the Plant
prototype property
```

```
Plant.prototype.amIOrganic = function () {
if (this.isOrganic)
console.log("I am organic, Baby!");
}
function Fruit (fruitName, fruitColor) {
this.name = fruitName;
this.color = fruitColor;
}
// Set the Fruit's prototype to Plant's constructor,
thus inheriting all of Plant.prototype methods and
properties.
Fruit.prototype = new Plant ();
// Creates a new object, aBanana, with the Fruit
constructor
var aBanana = new Fruit ("Banana", "Yellow");
// Here, aBanana uses the name property from
the aBanana object prototype, which is
Fruit.prototype:
console.log(aBanana.name); // Banana
// Uses the showNameAndColor method from the
Fruit object prototype, which is Plant.prototype.
The aBanana object inherits all the properties and
methods from both the Plant and Fruit functions.
console.log(aBanana.showNameAndColor()); // I
am a Banana and my color is yellow.
```

From <https://web.archive.org/web/20200513181548/https://javascriptissexy.com/javascript-prototype-in-plain-detailed-language/>

Prototype Attribute: Accessing Properties on Objects

Prototype is also important for accessing properties and methods of objects. The prototype attribute (or prototype object) of any object is the "parent" object where the inherited properties were originally defined.This is loosely analogous to the way you might inherit your surname from your father—he is your "prototype parent." If we wanted to find out where your surname came from, we would first check to see if you created it yourself; if not, the search will move to your prototype parent to see if you inherited it from him. If it was not created by him, the search continues to his father (your father's prototype parent). Similarly, if you want to access a property of an object, the search for the property begins directly on the object. If the JS runtime can't find the property there, it then looks for the property on the object's prototype—the object it inherited its properties from.

If the property is not found on the object's prototype, the search for the property then moves to prototype of the object's prototype (the father of the object's father—the grandfather). And this continues until there is no more prototype (no more great-grand father; no more lineage to follow). This in essence is the prototype chain: the chain from an object's prototype to its prototype's prototype and onwards. And JavaScript uses this prototype chain to look for properties and methods of an object.

If the property does not exist on any of the object's prototype in its prototype chain, then the property does not exist and *undefined* is returned.

This prototype chain mechanism is essentially the same concept we have discussed above with the prototype-based inheritance, except we are now focusing specifically on how JavaScript accesses object properties and methods via the prototype object.

This example demonstrates the prototype chain of an object's prototype object:

```
var myFriends = {name: "Pete"};

// To find the name property below, the search will begin directly on the myFriends object and will immediately find the name property because we defined the property name on the myFriend object. This could be thought of as a prototype chain with one link.
console.log(myFriends.name);

// In this example, the search for the toString () method will also begin on the myFriends' object, but because we never created a toString method on the myFriends object, the compiler will then search for it on the myFriends prototype (the object which it inherited its properties from).

// And since all objects created with the object literal inherits from Object.prototype, the toString method will be found on Object.prototype—see important note below for all properties inherited from Object.prototype.

myFriends.toString ();
```

Object.prototype Properties Inherited by all Objects

All objects in JavaScript inherit properties and methods from Object.prototype. These inherited properties and methods are *constructor, hasOwnProperty (), isPrototypeOf (), propertyIsEnumerable (), toLocaleString (), toString (), and valueOf ()*. ECMAScript 5 also adds 4 accessor methods to Object.prototype.

Here is another example of the prototype chain:

```
function People () {
this.superstar = "Michael Jackson";
}
// Define "athlete" property on the People prototype so that "athlete" is accessible by all objects that use the People () constructor.
People.prototype.athlete = "Tiger Woods";

var famousPerson = new People ();
famousPerson.superstar = "Steve Jobs";

// The search for superstar will first look for the superstar property on the famousPerson object, and since we defined it there, that is the property that will be used. Because we have overwritten the famousPerson's superstar property with one directly on the famousPerson object, the search will NOT proceed up the prototype chain.
console.log (famousPerson.superstar); // Steve Jobs

// Note that in ECMAScript 5 you can set a property to read only, and in that case you cannot overwrite it as we just did.
```

```
// This will show the property from the famousPerson prototype (People.prototype), since the athlete property was not defined on the famousPerson object itself.
console.log (famousPerson.athlete); // Tiger Woods

// In this example, the search proceeds up the prototype chain and find the toString method on Object.prototype, from which the Fruit object inherited—all objects ultimately inherits from Object.prototype as we have noted before.
console.log (famousPerson.toString()); // [object Object]
```

All built-in constructors (Array (), Number (), String (), etc.) were created from the Object constructor, and as such their prototype is Object.prototype.

From <https://web.archive.org/web/20200513181548/https://javascriptissexy.com/javascript-prototype-in-plain-detailed-language/>

```
function Student(name, grade) {
  this.name = name
  this.grade = grade
}

Student.prototype.sayName = function() {
  console.log(this.name)
}
Student.prototype.goToProm = function() {
  console.log("Eh.. go to prom?")
}
```

If you're using constructors to make your objects it is best to define functions on the prototype of that object. Doing so means that a single instance of each function will be shared between all of the Student objects. If we declare the function directly in the constructor, like we did when they were first introduced, that function would be duplicated every time a new Student is created. In this example, that wouldn't really matter much, but in a project that is creating thousands of objects, it really can make a difference.

From <https://www.theodinproject.com/lessons/node-path-javascript-objects-and-object-constructors>

```
function Student() {
}

Student.prototype.sayName = function() {
  console.log(this.name)
}

function EighthGrader(name) {
  this.name = name
  this.grade = 8
}

EighthGrader.prototype = Object.create(Student.prototype)

const carl = new EighthGrader("carl")
carl.sayName() // console.logs "carl"
carl.grade // 8
```

A warning... this doesn't work:

EighthGrader.prototype = Student.prototype

because it will literally set EighthGrader's prototype to Student.prototype (i.e. not a copy), which could cause problems if you want to edit something in the future. Consider one more example:

```
function Student() {
}

Student.prototype.sayName = function() {
  console.log(this.name)
}

function EighthGrader(name) {
  this.name = name
  this.grade = 8
}

// don't do this!!!
EighthGrader.prototype = Student.prototype

function NinthGrader(name) {
  this.name = name
  this.grade = 9
}

// noooo! not again!
NinthGrader.prototype = Student.prototype

NinthGrader.prototype.sayName = function() {console.log("HAHAHAHAHAHA")}

const carl = new EighthGrader("carl")
carl.sayName() //uh oh! this logs "HAHAHAHAHAHA" because we edited the sayName
```

- Here we have two solutions for this

```
function Plant() {
    this.country = "mexico";
    this.isOrganic = true;
}

Plant.prototype.amIOrganic = function () {
    if (this.isOrganic) {
        return "I am Organic";
    }
    else {
        return "I am not Organic";
    }
}

function Fruit(name) {
    this.name = name;
}

Fruit.prototype = new Plant();
var apple = new Fruit("Apple");

console.log(apple.amIOrganic());
```

```
function Plant() {
    this.country = "mexico";
    this.isOrganic = true;
}

Plant.prototype.amIOrganic = function () {
    if (this.isOrganic) {
        return "I am Organic";
    }
    else {
        return "I am not Organic";
    }
}

function Fruit(name) {
    this.name = name;
}

Fruit.prototype = Object.create(Plant.prototype);

var apple = new Fruit("Apple");

console.log(apple.amIOrganic());
```