**Difference between HTTP1.1 vs HTTP2**

HTTP2 improved in a number of ways that allowed for faster content delivery and improved user experience, some of them are like:

* **Binary protocols** – Binary protocols **consume less bandwidth** & more efficiently parsed than the textual protocols used by HTTP1.1. Additionally, it can better handle elements such as whitespace, capitalization and line endings.
* **Multiplexing** – HTTP2 is **multiplexed**, i.e. it can initiate multiple requests in parallel over a single TCP connection. As a result, web pages containing several elements are delivered over one TCP connection. These capabilities solve the problem of head of line blocking in HTTP1.1.
* **Header compression** – HTTP2 uses **header compression** to reduce the overhead caused by TCP’s slow start mechanism.
* **Server push** – HTTP2 servers push likely to be **used** resources into a browser’s **cache**, even before they’re requested. This allows browsers to display content without additional request cycles.
* **Increased security** – Web browsers only support HTTP2 via **encrypted** **connections**, increasing user and application security.

**Objects & its internal representation**

**in JavaScript**

Objects, in JavaScript, is it’s most important data-type and forms the building blocks for modern JavaScript. These objects are quite different from JavaScript’s primitive data-types (Number, String, Boolean, null, undefined and symbol) in the sense that while these primitive data-types all store a single value each (depending on their types).

Objects are more complex and each object may contain any combination of these primitive data-types as well as reference data-types.

An object is a reference data type. Variables that are assigned a reference value are given a reference or a pointer to that value. That reference or pointer points to the location in memory where the object is stored. The variables don’t actually store the value.

Objects in JS may be defined as an unordered collection of related data, of primitive or reference types, in the form of “key: value” pairs. These keys can be variables or functions and are called properties and methods, respectively, in the context of an object.  
An object can be created with figure brackets {…} with an optional list of properties. A property is a “key: value” pair, where a key is a string (also called a “property name”), and value can be anything.

An example of a JS Object :

**let student = {  
name : “Divyesh Bhuva”,  
course : “MERN”,  
year : “2022”  
}**

In the above example “name”, “course”, “year” are “keys” and “Divyesh Bhuva”, “MERN” and “2022” are values of these keys respectively.

Each of these keys is referred to as properties of the object. An object in JavaScript may also have a function as a member, in which case it will be known as a method of that object.

Let’s see with an example :  
// Demonstrating a simple object in JS  
**let student = {  
name : “Divyesh Bhuva”,  
course : “MERN”,  
year : “2022”  
}**

displayInfo : function(){  
console.log(${student.name} is started learning ${student.course} in ${student.year});  
}  
student.displayInfo();

Output:  
In the above example, “displayinfo” is a method of the student object that is being used to work with the object’s data, stored in its properties.

**Properties of JavaScript Object**:  
The property names can be strings or numbers. In case the property names are numbers, they must be accessed using the “bracket notation” like this :  
**let student = {  
name : “Divyesh Bhuva”,  
course : “MERN”,  
year : “2022”  
}**

displayInfo : function(){  
console.log(The value of the key year is ${studentl[‘year’]});  
}  
  
student.displayInfo();

Output:  
But more on the bracket notation later. Property names can also be strings with more than one space separated words. In which case, these property names must be enclosed in.

**Creating Objects**:There are several ways or syntax’s to create objects. One of which, known as the Object literal syntax, we have already used. Besides the object literal syntax, objects in JS may also be created using the constructors, Object constructor or the prototype pattern.

Using the **Object literal syntax** : Object literal syntax uses the {…} notation to initialize an object an its methods/properties directly.

Let’s see an example of creating objects with this method :  
**var obj = {  
member1 : value1,  
member2 : value2,  
};**

These members can be anything — strings, numbers, functions, arrays or even other objects. An object like this is referred to as an object literal. This is different from other methods of object creation which involve using constructors and classes or prototypes, which have been discussed below.

**Object Constructor**: Another way to create objects in JavaScript involves using the “Object” constructor. The Object constructor creates an object wrapper for the given value. This, used in conjunction with the “new” keyword allows us to initialize new objects.

Example :  
**const student = new Object();  
student.name = ‘DIVYESH BHUVA;  
student.course = ‘MERN;  
student.year = 2022;**

student.displayInfo = function(){  
console.log(${student.name} is started learning ${student.course} in ${student.year});  
}

student.displayInfo();

Output:  
The two methods mentioned above are not well suited to programs that require the creation of multiple objects of the same kind, as it would involve repeatedly writing the above lines of code for each such object. To deal with this problem, we can make use of two other methods of object creation in JavaScript that reduces this burden significantly, as mentioned below:  
Constructors: Constructors in JavaScript, like in most other OOP languages, provides a template for creation of objects. In other words, it defines a set of properties and methods that would be common to all objects initialized using the constructor.

Let us see an example :  
function Vehicle(name, maker) {  
this.name = name;  
this.maker = maker;  
}

let car1 = new Vehicle(‘Veron, ‘Bugati’);  
let car2 = new Vehicle(‘Gallaro’, ‘Lamborghini’)

console.log(car1.name); // Output: Veron  
console.log(car2.name); // Output: Gallaro

Output:  
Notice the usage of the “new” keyword before the function Vehicle. Using the “new” keyword in this manner before any function turns it into a constructor. What the “new Vehicle()” actually does is :  
It creates a new object and sets the constructor property of the object to students (It is important to note that this property is a special default property that is not enumerable and cannot be changed by setting a “constructor: someFunction” property manually).

Then, it sets up the object to work with the Vehicle function’s prototype object (Each function in JS gets a prototype object, which is initially just an empty object but can be modified. The object, when instantiated inherits all properties from its constructor’s prototype object).

Then calls Vehicle() in the context of the new object, which means that when the “this” keyword is encountered in the constructor(vehicle()), it refers to the new object that was created in the first step. Once this is finished, the newly created object is returned to car1 and car2(in the above example).

Inside classes, there can be special methods named constructor().

class staff {  
constructor()  
{  
this.name = “Dev”;  
}  
}

let employee1 = new people();

// Output : Dev  
console.log(employee1.name);

Output:  
Having more than one function in a class with the name of constructor() results in an error.

**Prototypes**: Another way to create objects involves using prototypes. Every JS function has a prototype object property by default (it is empty by default). Methods or properties may be attached to this property.

The basic syntax used as below:  
let obj = Object.create(prototype\_object, propertiesObject)  
// the second propertiesObject argument is optional  
An example of making use of the Object.create() method is:  
let footballers = {  
position: “Defender”  
}

let footballer1 = Object.create(footballers);

// Output : Defender  
console.log(footballer1.position);

Output: In the above example footballers served as a prototype for creating the object “footballer1”.

All objects created in this way inherits all properties and methods from its prototype objects. Prototypes can have prototypes and those can have prototypes and so on. This is referred to as prototype chaining in JavaScript. This chain terminates with the Object.prototype which is the default prototype fallback for all objects. JS objects, by default, inherit properties and methods from Object.prototype but these may easily be overridden. It is also interesting to note that the default prototype is not always Object.prototype. For example Strings and Arrays have their own default prototypes — String.prototype and Array.prototype respectively.

Accessing Object Members  
Object members(properties or methods) can be accessed using the  
dot notation :  
(objectName.memberName)  
let student = {  
name : “Divyesh”,  
course : “MERN”,  
year : 2022,  
displayinfo : function() {  
console.log(${student.name} is started learning ${student.course} in ${student.year});}

}

console.log(student.name);

console.log(student.year);

Output:

Bracket Notation :  
objectName[“memberName”]  
let student = {  
name : “Divyesh Bhuva”,  
course : “MERN”,  
year : 2022,  
displayinfo : function() {  
document.write(${student.name} is started learning ${student.course} in ${student.year});  
}  
}

// Output : Divyesh Bhuva  
console.log(student[‘name’]);

// Output: 2022  
console.log(student[‘year]);

Output: Unlike the dot notation, the bracket keyword works with any string combination, but not limited to multi-word strings.

For example:  
Employee.first name // invalid  
Employee[“first name”] // valid  
Unlike the dot notation, the bracket notation can also contain names which are results of any expressions variables whose values are computed at run-time.  
For instance :  
let key = “first name” Employee[key] = “Name Surname”  
Similar operations are not possible while using the dot notation.