Q.1) List 5 difference between Browser JS (console) v Nodejs

>In the browser, most of the time what you are doing is interacting with the DOM, or other Web Platform APIs like Cookies. Those do not exist in Node.js, you don't have the document, window and all the other objects that are provided by the browser.

>And in the browser, we don't have all the nice APIs that Node.js provides through its modules, like the filesystem access functionality.

>Another big difference is that in Node.js you control the environment. Unless you are building an open source application that anyone can deploy anywhere, you know which version of Node.js you will run the application on. Compared to the browser environment, where you don't get the luxury to choose what browser your visitors will use, this is very convenient.

>Since JavaScript moves so fast, but browsers can be a bit slow to upgrade, sometimes on the web you are stuck with using older JavaScript / ECMAScript releases.

>You can use Babel to transform your code to be ES5-compatible before shipping it to the browser, but in Node.js, you won't need that.

>Another difference is that Node.js supports both the CommonJS and ES module systems (since Node.js v12), while in the browser we are starting to see the ES Modules standard being implemented.

>In practice, this means that you can use both require() and import in Node.js, while you are limited to import in the browser.

>Building apps that run in the browser is a completely different thing than building a Node.js application.

>Both the browser and Node.js use JavaScript as their programming language. Despite the fact that it's always JavaScript, there are some key differences that make the experience radically different.

> “location” object is related to a particular url; that means it is for page specific. So, node doesn’t require that.

>Node has “global”, which is a predefined global object. It contains several functions that are not available in browsers, because they are needed for server side works only.

>“require” object is predefined in Node which is used to include modules in the app.

In Node everything is a module. You must keep your code inside a module.

As both of them are JavaScript executor, and Node uses the JavaScript engine of a browser (Chrome), so differences are not much there. It is just the Node wrapper which has been written on top of JavaScript V8 Runtime engine, which is deleting few objects and also including some according to the requirement of Node.

Q.2) Watch & summary 5 points –

Following are steps involved in web page rendering:

1. Construction of DOM:

* Browser receives html page in binary format from server.
* When the browser reads the HTML document, whenever it encounters an HTML element, it creates a JS object called a **Node**. Eventually, all html elements will be converted to a **Node**.
* After the browser has created nodes from the HTML document, it has to create a "tree-like" structure of these node objects.

### 2. Construction of CSSOM

* After constructing the DOM, the browser reads CSS from all the sources & constructs a CSSOM (CSS Object Model) - a tree-like structure.
* Each node in this tree contains CSS-style information that will be copied to the DOM element it targets.
* Most of the browser comes with their own stylesheet which is called **user-agent stylesheets**. It is a default stylesheet used by web browsers. in the absence of any CSS applied, the browser still has to render the content somehow, and the browser uses the user-agent stylesheet for that.

### 3. Construction of Render Tree

* DOM & CSSOM are combined together to form a Render tree that contains the nodes which have to be displayed on the page.
* From the root of the DOM tree, each **visible** node is traversed and a respective CSSOM rule is applied. Finally, it gives the render tree containing **visible** nodes with content and styling.
* It is a low-level representation of what will eventually get printed on the screen, it won't contain nodes that do not hold any area in the pixel matrix (like head, meta, link tags).
* 4. This phase can be said as a **geometry phase**, where we calculate the geometry of the nodes.
* In the layout phase, the exact position of the nodes and their size respective to the view-port of the browser is computed. In this way, a **box model** is generated which knows the exact positions and size. This process is also known as **layout** or **reflow**.

### 5. Painting Phase

* As we know the visible nodes, their styling, & their geometry, now all this information is used to render the nodes from the render tree to actual pixels on the screen. This process is referred to as Painting. It uses the UI backend layer.

Q.3)Read

> I have read it.

Q.4) Execute the below code and write your description in txt file

=> typeof.txt file uploaded

Q.5) What is prototype?

Here, we will have a look at “prototype in designing” and “prototype in javascript”.

**Prototype in designing:**

A prototype is an early sample, model or release of a product created to test a concept or process. Typically, a prototype is used to test a new design to improve the accuracy of analysts and system users. It is the step between the formalization and the evaluation of an idea.

Prototypes **are a crucial part of the design process** and a practice used in all design disciplines. From architects, engineers, industrial designers and even service designers, they make their prototypes to test their designs before investing in their mass production.

Prototypes often fail when tested, and this shows designers where the defects are and sends the team to refine or repeat the proposed solutions based on real user feedback. Another advantage of prototyping is that, because the investment is small, the risk is low.

**Types of prototype:**

1.Low fidelity prototypes

2.Medium fidelity prototypes

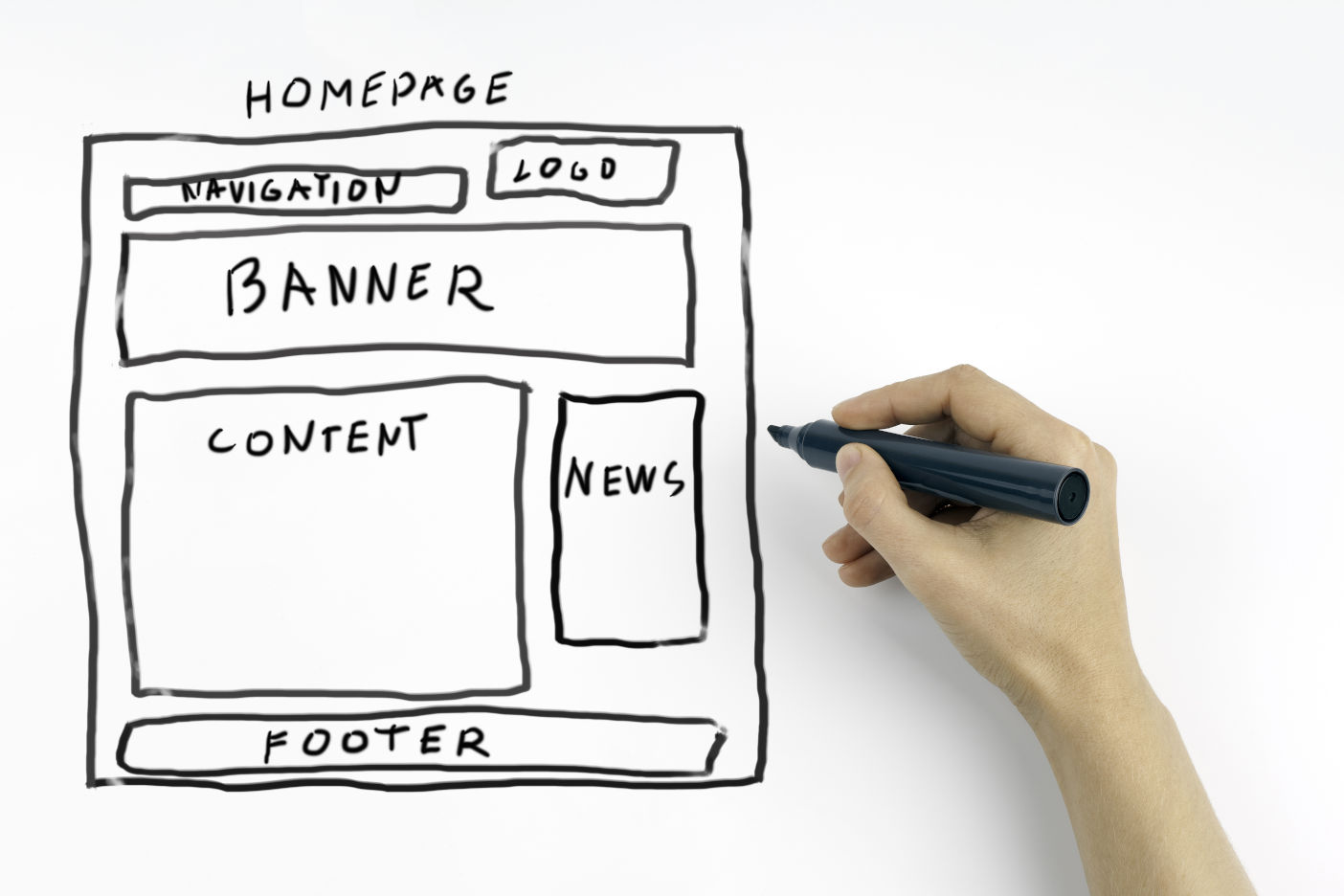
3. Hi-fi prototypes

**1.Low fidelity prototypes**

A low-fidelity prototype is a low-tech and straightforward type of prototype. This prototype is basically not close to the real product. In the classification of prototype fidelity, the closer the prototype is to the final product, the higher the fidelity.

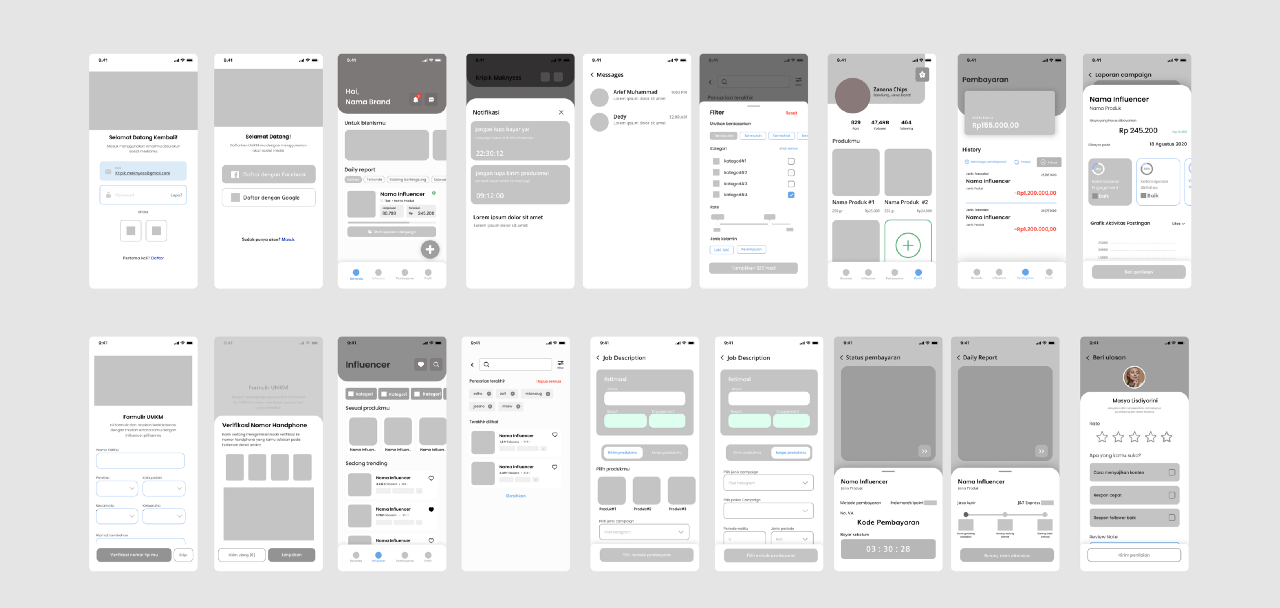
For low fidelity, it means that several aspects of the final product are not captured. These elements include visuals, content, and interactivity. A low-fidelity has only a few visual attributes of the final product presented. This is also true for the content. Only the necessary content is included, while other details are excluded from making it easy and fast to design.

It consist of sticky notes and sketches (thus a paper prototypes with the name), which is ideal for brainstorming and high-level collaboration.



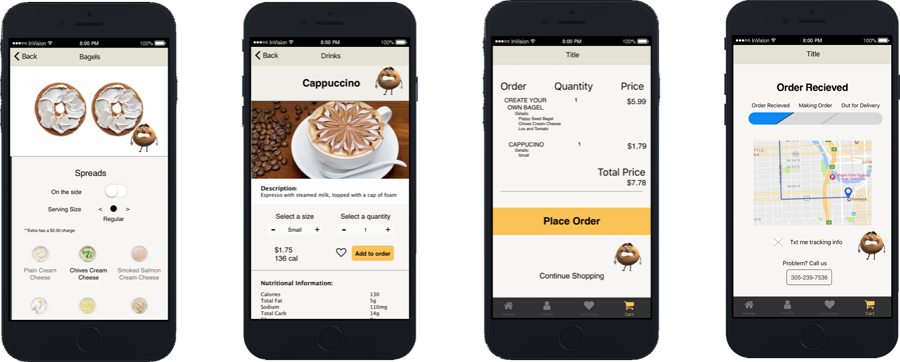
**2.Medium fidelity prototypes**

**They** are often called wireframes, and are often made in black and white, limiting the design approach to user flows and information architecture.



**3. Hi-fi prototypes**

The high-fidelity prototype - known as **high-tech, high-fi or hi-fi prototype**, is a comprehensive and interactive prototype that is quite close to the final products with lots of functions, interactions and details. It’s often used in the later usability evaluation to discover the potential issues that a web/app design may still exist.



**Prototype in javascript:**

**Object prototype:**

Prototypes are the mechanism by which JavaScript objects inherit features from one another. All JavaScript objects inherit properties and methods from a prototype.

Here we will explain what a prototype is, how prototype chains work, and how a prototype for an object can be set.

**Prototype chain:**

const myObject = {

city: 'Madrid',

greet() {

console.log(`Greetings from ${this.city}`);

}

}

myObject.greet(); // Greetings from Madrid

This is an object with one data property, city, and one method, greet(). If you type the object's name followed by --proto-- into the console, like myObject.\_\_proto\_\_, then the console will pop up a list of all the properties available to this object. You'll see that as well as city and greet, there are lots of other properties!

1. \_\_**constructor**: *ƒ Object()*
2. **hasOwnProperty**: *ƒ hasOwnProperty()*
3. **isPrototypeOf**: *ƒ isPrototypeOf()*
4. **propertyIsEnumerable**: *ƒ propertyIsEnumerable()*
5. **toLocaleString**: *ƒ toLocaleString()*
6. **toString**: *ƒ toString()*
7. **valueOf**: *ƒ valueOf()*
8. **\_\_defineGetter\_\_**: *ƒ \_\_defineGetter\_\_()*
9. **\_\_defineSetter\_\_**: *ƒ \_\_defineSetter\_\_()*
10. **\_\_lookupGetter\_\_**: *ƒ \_\_lookupGetter\_\_()*
11. **\_\_lookupSetter\_\_**: *ƒ \_\_lookupSetter\_\_()*
12. **\_\_proto\_\_**: (...)
13. get \_\_proto\_\_: *ƒ \_\_proto\_\_()*
14. set \_\_proto\_\_: *ƒ \_\_proto\_\_()*

Try accessing one of them:

myObject.toString(); // "[object Object]"

It works (even if it's not obvious what toString() does).

What are these extra properties, and where do they come from?

Every object in JavaScript has a built-in property, which is called its **prototype**. The prototype is itself an object, so the prototype will have its own prototype, making what's called a **prototype chain**. The chain ends when we reach a prototype that has null for its own prototype.

**Note:** The property of an object that points to its prototype is **not** called prototype. Its name is not standard, but in practice all browsers use [\_\_proto\_\_](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/proto). The standard way to access an object's prototype is the [Object.getPrototypeOf()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getPrototypeOf) method.

When you try to access a property of an object: if the property can't be found in the object itself, the prototype is searched for the property. If the property still can't be found, then the prototype's prototype is searched, and so on until either the property is found, or the end of the chain is reached, in which case undefined is returned.

So when we call myObject.toString(), the browser:

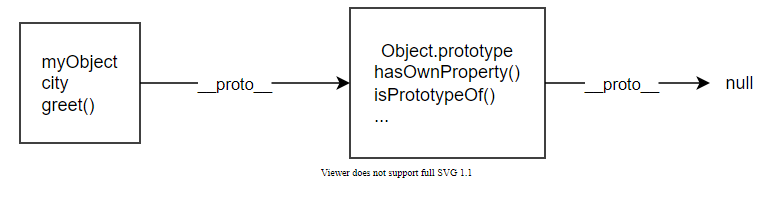
* looks for toString in myObject
* can't find it there, so looks in the prototype object of myObject for toString
* finds it there, and calls it.

What is the prototype for myObject? To find out, we can use the function Object.getPrototypeOf():

Object.getPrototypeOf(myObject);

// {constructor: ƒ, \_\_defineGetter\_\_: ƒ, \_\_defineSetter\_\_: ƒ, hasOwnProperty: ƒ, \_\_lookupGetter\_\_: ƒ, …}

This is an object called Object.prototype, and it is the most basic prototype, that all objects have by default. The prototype of Object.prototype is null, so it's at the end of the prototype chain:



The prototype of an object is not always Object.prototype. Try this:

const myDate = new Date();

let object = myDate;

do {

object = Object.getPrototypeOf(object);

console.log(object);

} while (object);

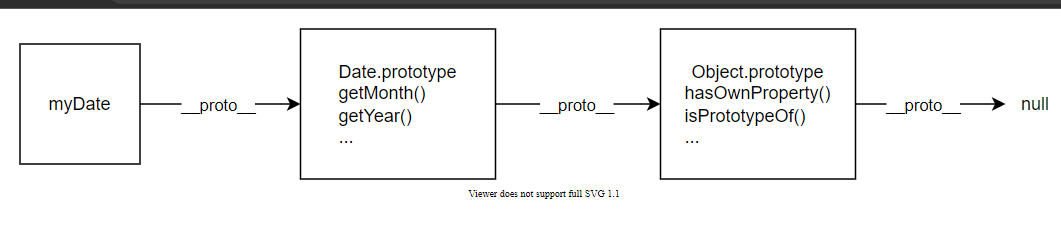
// Date.prototype

//{constructor: ƒ, toString: ƒ, toDateString: ƒ, toTimeString: ƒ, toISOString: ƒ, …}

//{constructor: ƒ, \_\_defineGetter\_\_: ƒ, \_\_defineSetter\_\_: ƒ, hasOwnProperty: ƒ, \_\_lookupGetter\_\_: ƒ, …}

//null

This code creates a Date object, then walks up the prototype chain, logging the prototypes. It shows us that the prototype of myDate is a Date.prototype object, and the prototype of that is Object.prototype.



In fact, when you call familiar methods, like myDate2.getMonth(), you are calling a method that's defined on Date.prototype.

## [Shadowing properties](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes#shadowing_properties)

What happens if you define a property in an object, when a property with the same name is defined in the object's prototype? Let's see:

const myDate = new Date(1995, 11, 17);

console.log(myDate.getYear()); // 95

myDate.getYear = function() {

console.log('something else!')

};

console.log(myDate.getYear()); // 'something else!'

This should be predictable, given the description of the prototype chain. When we call getYear() the browser first looks in myDate for a property with that name, and only checks the prototype if myDate does not define it. So when we add getYear() to myDate, then the version in myDate is called.

This is called "shadowing" the property.

## [Setting a prototype](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes#setting_a_prototype)

There are various ways of setting an object's prototype in JavaScript, and here we'll describe two: Object.create() and constructors.

### [Using Object.create](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes#using_object.create)

The Object.create() method creates a new object and allows you to specify an object that will be used as the new object's prototype.

Here's an example:

const personPrototype = {

greet() {

console.log('hello!');

}

}

const carl = Object.create(personPrototype);

carl.greet(); // hello!

Here we create an object personPrototype, which has a greet() method. We then use Object.create() to create a new object with personPrototype as its prototype. Now we can call greet() on the new object, and the prototype provides its implementation.

### [Using a constructor](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes#using_a_constructor)

In JavaScript, all functions have a property named prototype. When you call a function as a constructor, this property is set as the prototype of the newly constructed object (by convention, in the property named \_\_proto\_\_).

So if we set the prototype of a constructor, we can ensure that all objects created with that constructor are given that prototype:

const personPrototype = {

greet() {

console.log(`hello, my name is ${this.name}!`);

}

}

function Person(name) {

this.name = name;

}

Person.prototype = personPrototype;

Person.prototype.constructor = Person;

Here we create:

* an object personPrototype, which has a greet() method
* a Person() constructor function which initializes the name of the person to create.

We then set the Person function's prototype property to point to personPrototype.

The last line (Person.prototype.constructor = Person;) sets the prototype's constructor property to the function used to create Person objects. This is required because after setting Person.prototype = personPrototype; the property points to the constructor for the personPrototype, which is Object rather than Person (because personPrototype was constructed as an object literal).

After this code, objects created using Person() will get personPrototype as their prototype.

const reuben = new Person('Reuben');

reuben.greet(); // hello, my name is Reuben!

This also explains why we said earlier that the prototype of myDate is called Date.prototype: it's the prototype property of the Date constructor.

### [Own properties](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes#own_properties)

The objects we create using the Person constructor above have two properties:

* a name property, which is set in the constructor, so it appears directly on Person objects
* a greet() method, which is set in the prototype.

It's common to see this pattern, in which methods are defined on the prototype, but data properties are defined in the constructor. That's because methods are usually the same for every object we create, while we often want each object to have its own value for its data properties (just as here where every person has a different name).

Properties that are defined directly in the object, like name here, are called **own properties**, and you can check whether a property is an own property using the static [Object.hasOwn()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn) method:

const irma = new Person('Irma');

console.log(Object.hasOwn(irma, 'name')); // true

console.log(Object.hasOwn(irma, 'greet')); // false

## [Prototypes and inheritance](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes#prototypes_and_inheritance)

Prototypes are a powerful and very flexible feature of JavaScript, making it possible to reuse code and combine objects.

In particular they support a version of **inheritance**. Inheritance is a feature of object-oriented programming languages that lets programmers express the idea that some objects in a system are more specialized versions of other objects.

For example, if we're modeling a school, we might have professors and students: they are both people, so have some features in common (for example, they both have names), but each might add extra features (for example, professors have a subject that they teach), or might implement the same feature in different ways. In an OOP system we might say that professors and students both **inherit from** people.

You can see how in JavaScript, if Professor and Student objects can have Person prototypes, then they can inherit the common properties, while adding and redefining those properties which need to differ.