Rendering of websites in browser

Rendering of websites means taking the request from the browser to the server and sending back the required content. Rendering displays what user sees on the screen. It takes the HTML/CSS code from the server and displays it to the user.

PARSER

**HTML**

PARSER

**CSS**

ATTACHMENT

Parsing HTML :- HTML is a heirarchial structure. It is used to display the contents of the web page to the user. HTML has various tags and attributes. Html begins with HTML tag. Further it has Body tag. These HTML tags are parsed and turned into a DOM tree.

Parsing CSS :- CSS is used to give styling to the html page. The css styling is parsed with the DOM tree and the render tree is created.

ATTACHMENT:- Once the render tree is created the rendering engine place each and every html tags on the screen with the images or various attributes.

Painting:- This results in how a HTML page must look. The rendering engine understands the elements that are part of web page and place them accordingly giving the proper user interface.

RENDERING OBJECT:- The content of the page is been manipulated by the server. The engine takes the request from the user and pass it to the server and goes back to the browser with the content.

**List of Javascript engines:-**

* Chrome’s V8 engine:- Lars Bak.
* Chakra core:- Microsoft.
* Spidermonkey :- Brendan Eich.
* Rhino :- Mozilla foundation
* Nashorn:-
* Jerryscript:-
* Safari :- JS core.

Q2) Difference between interpreted and compiled language.

In compiled language the original program is translated into machine language instructions,which are executed by hardware. In compiled language , the original program is translated into binary instructions for a virtual machine, i.e. bytecode. In an interpreted implementation of language the source cod eis not run directly by target machine. Another programs reads and then implement the source code. This other program is known as interpretor. Computers cannot actually run the code that you write javascript (or any other language for that matter). Computers can only run machine code. The machine code that a particular computer can run is defined within the processor that is going to run those commands and can be different for different processors.

Q3) What are different ways of defining variables in js?

Variables are the containers to store the data.

Var:-

Var x=10;

Here, the variable x is equal to number 10 and the value in stored in var .

Function begin(){

For(var i=0;i<5;i++)

{

Console.log(i);

}console.log(i);

}

In the above code the variable I is declared inside the for loop. What is strange here is that, although the variable ***i*** is declared inside the for-loop, it is still accessible outside the scope of the for-loop. This is because the ***var*** variables are accessible within the scope of the function that they are declared.

The variable ***i*** is accessible anywhere within the function begin(). This is a behaviour that is unique to the JavaScript ***var*** keyword. In many other languages, the expected behaviour, would be that the scope of ***i*** is limited to the block within which it is declared.

Let:-

IF the variables are to be scoped within a block of code. If we declare a variable within a block of code say a for-loop, we want it to live within that block and be inaccessible outside of it. This behavior can be achieved in JavaScript using the ***let*** keyword.

Let’s go back to our previous example, and replace the ***var*** with ***let*** instead.

Function begin(){

For(let i=0;i<5;i++)

{

Console.log(i);

}console.log(i);

}

Here the number 5 will not be printed as the scope is only till the same block.

Const:-

Const is used to declare any value which is not going to be changed anywhere within the code.

Const name=pallavi;

The name pallavi is not going to change, it is the constant value. In such cases const is used.

Q4) What are different types of scope in javascript?

Scope in JavaScript defines accessibility of variables, objects and functions.

There are two types of scope in JavaScript.

1. Global scope
2. Local scope

Global Scope

Variables declared outside of any function become global variables. Global variables can be accessed and modified from any function. When we try to access a variable in our code, the interpreter first looks for it in the current scope. If it wasn’t found, it goes up through the chain of scopes. At the very top of it, there is the **global scope**. Variables declared there can be both accessed and altered anywhere in the code. **Global scope** exists as long as your application runs and there is only one instance of it. There is another type of scope called a **local scope**that can enclose the variables that are declared inside of it.

Q5) What are different types of function ?

A **function** is a parametric block of code defined one time and called any number of times later. The function code has a slight impact on what declaration type to choose. Important is how the function interacts with the external components (the outer scope, the enclosing context, object that owns the method, etc) and the invocation type (regular function invocation, method invocation, constructor call, etc).

For instance you need this  on a function invocation to be the same as the enclosing context (i.e. inherits this from the outer function). The best option is to use an arrow function, which provides the necessary context transparency.

1.Function declaration

function *name*([*param*[, *param*[, ... *param*]]]) {

*statements*

}

2.Function expressions

A function expression is similar to and has the same syntax as a function declaration function expression

 for details). A function expression may be a part of a larger expression. One can define "named" function expressions (where the name of the expression might be used in the call stack for example) or "anonymous" function expressions. Function expressions are not hoisted onto the beginning of the scope, therefore they cannot be used before they appear in the code.

function [*name*]([*param*[, *param*[, ... *param*]]]) {

*statements*

}

Q6.What are different types of scope in javascript?

Node.js is a server-side platform built on Google Chrome's JavaScript Engine (V8 Engine). Node.js was developed by Ryan Dahl in 2009Pallavi: Node.js is an open source, cross-platform runtime environment for developing server-side and networking applications. Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows, and Linux.Node.js also provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js to a great extent.

Following are the areas where Node.js is proving itself as a perfect technology partner.

I/O bound Applications

Data Streaming Applications

Data Intensive Real-time Applications (DIRT)

JSON APIs based Applications

Single Page Applications

JavaScript is a single-threaded language. JavaScript has a single call stack in which it keeps track of what function we’re currently executing and what function is to be executed after that.When you’re about to execute a function it is added on the call stack.But when we make a request or put a timeout on something, every time you call a setTimeout function or you do some async operation — it is added to the Event Table. The Event Queue is a data structure similar to the stack — again you add items to the back but can only remove them from the front. It kind of stores the correct order in which the functions should be executed. Here Event Loop works. This is a constantly running process that checks if the call stack is empty. Imagine it like a clock and every time it ticks it looks at the Call Stack and if it is empty it looks into the Event Queue. If there is something in the event queue that is waiting it is moved to the call stack. If not, then nothing happens.

Q7)What is promise ? How do you create a promise ?

Promises are used to handle asynchronous operations in JavaScript. They are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code. Prior to promises events and callback functions were used but they had limited functionalities and created unmanageable code. Multiple callback functions would create callback hell that leads to unmanageable code. Events were not good at handling asynchronous operations. Promises are the ideal choice for handling asynchronous operations in the simplest manner. They can handle multiple asynchronous operations easily and provide better error handling than callbacks and events.

var promise = new Promise(function(resolve, reject) {

const x = "apple";

const y = "apple"

if(x === y) {

resolve();

} else {

reject();

}

});

promise. then(function () {

console.log('This is an APPLE');

}). catch(function () {

console.log('Some error has occured');

});