Subject : Internet Programming

Written Assignment No. 1

1. **Compare XML and JSON.** CO1 **Ans:**

Aspect	**XML**	**JSON**
Format	Uses tags enclosed in angle brackets (<>)	Uses key-value pairs in curly braces ({})
Human Readability	Contains explicit opening and closing tags, making it more verbose	Employs a simpler, more compact syntax
Data Types	Supports various data types (string, number, boolean, etc.)	Primarily deals with strings, numbers, booleans, arrays, and objects
Namespaces	Supports namespaces for avoiding naming conflicts	Lacks native support for namespaces
Attribute	Allows attributes to be attached to elements	Doesn't natively support attributes
Whitespace Handling	Preserves whitespace, leading to larger file sizes	Generally ignores whitespace, resulting in smaller files
Schema	Can be validated against Document Type Definitions (DTD) or XML Schema	Lacks built-in validation, but can use JSON Schema for validation

2. Explain different types of arrow functions. CO2 **Ans :**

- Arrow functions are a concise and modern feature introduced in JavaScript (ECMAScript 6) that provides an alternative way to define functions. They offer a more streamlined syntax compared to traditional function expressions, making code cleaner and more readable.

1. Basic Arrow Functions :

The simplest form of an arrow function takes one or more parameters and an expression that is evaluated and returned.

Example : const add = (a, b) => a + b;

2. Arrow Function with Multiple Statements :

You can use curly braces to write multiple statements inside an arrow function. Remember to explicitly return a value if using multiple statements.

Example :
```
const multiply = (a, b) => {
    const result = a * b;
    return result;
    };
```

3. Arrow Function with No Parameters:

If your function doesn't require any parameters, you still need to include empty parentheses.

Example : const greet = () => "Hello, World!";

4. Arrow Function with Single Parameter:

If your function takes a single parameter, you can omit the parentheses around the parameter.

Example : const square = num => num * num;

5. Arrow Functions with Default Parameters :

You can use default parameter values in arrow functions just like in regular functions.

Example : const greet = (name = "Stranger") => `Hello, ${name}!`;

6. Arrow Functions with Rest Parameters :

Arrow functions can use the rest parameter syntax to handle a variable number of arguments.

Example : const sum = (...numbers) => numbers.reduce((total, num) => total + num, 0);

3. What is DNS? Explain working of DNS. CO1 **Ans :**

All computers on the Internet that serve content for massive retail websites, find and communicate with one another by using numbers. • These numbers are known as IP addresses.

• When you open a web browser and go to a website, you don't have to remember and enter a long number. • Instead, you can enter a domain name like abc.com.

• A DNS service that translates human readable names like www.abc.com into the numeric IP addresses like 192.1.2.1 that computers use to connect to each other.

• The Internet's DNS system works much like a phone book by managing the mapping between names and numbers. DNS servers translate requests for names into IP addresses, controlling which server an end user will reach when they type a domain name into their web browser.

• These requests are called queries

In summary, DNS is a crucial system that enables users to access websites by translating domain names into IP addresses. It works through a network of servers and involves a hierarchical resolution process to provide accurate and efficient domain-to-IP mappings.

4. Explain Promises in ES6. C02 **Ans :**

1. Introduction to Promises: Promises are a feature introduced in ES6 (ECMAScript 2015) to handle asynchronous operations in a more organized and manageable way.

2. Asynchronous Operations: In JavaScript, certain operations take time to complete, such as fetching data from a server. Promises help manage these operations without blocking the main thread.

3. Three States of a Promise: A Promise can be in one of three states: pending (initial state), fulfilled (operation completed successfully), or rejected (operation failed).

4. Promise Creation: Promises are created using the `Promise` constructor. They take a function with two arguments: `resolve` (for success) and `reject` (for failure).

Example:

```
const fetchData = new Promise((resolve, reject) => {
// Fetch data from a server...
if (dataReceivedSuccessfully) {
resolve(data);
} else {
reject('Error fetching data');
}
});
```

5. `.then()` Method: The `.then()` method is used to handle the fulfillment of a Promise. It takes a function as an argument that gets executed when the Promise is resolved.

Example:

```
fetchData.then(data => {
console.log(data);
}).catch(error => {
console.error(error);
});
```

6. Chaining `.then()` Methods: `.then()` methods can be chained to perform a sequence of operations. Each `.then()` returns a new Promise.

Example:
```
fetchData.then(data => {
return processData(data);
}).then(processedData => {
console.log(processedData);
}).catch(error => {
console.error(error);
});
```

7. `.catch()` Method: The `.catch()` method is used to handle Promise rejections. It takes a function that executes when the Promise is rejected.

Example:
```
fetchData.catch(error => {
console.error(error);
});
```

8. `.finally()` Method: The `.finally()` method is used to execute code regardless of whether the Promise is resolved or rejected.

Example:
```
fetchData.finally(() => {
console.log('Fetching operation completed.');
});
```

9. Promisifying Functions: You can convert callback-based functions into Promise-based functions using utilities like `util.promisify` (Node.js).

Example:
```
const fs = require('fs');
const { promisify } = require('util');
const readFileAsync = promisify(fs.readFile);

readFileAsync('file.txt')
.then(data => {
console.log(data.toString());
})
```

```
.catch(error => {
console.error(error);
});
```

10. Parallel Promises with `Promise.all()`: `Promise.all()` takes an array of Promises and returns a new Promise that fulfills when all the input Promises are resolved.

Example:

```
const promise1 = fetchData1();
const promise2 = fetchData2();

Promise.all([promise1, promise2])
.then(results => {
console.log('Both operations completed:', results);
})
.catch(error => {
console.error(error);
});
```

11. Race with `Promise.race()`: `Promise.race()` returns a new Promise that fulfills or rejects as soon as any of the input Promises fulfills or rejects.

Example:

```
const timeoutPromise = new Promise((resolve, reject) => {
setTimeout(() => {
reject('Operation timed out');
}, 5000);
});

Promise.race([fetchData, timeoutPromise])
.then(data => {
console.log(data);
})
.catch(error => {
console.error(error);
```