# UNIT- II

| Unit -III | | COs | Hrs. | SEE Marks |
|---|---|---|---|---|
| Unit Title | XML & JSON | | | |
| FLOWR<br>Syntax of XML, document structure, and document type definition, | | 3 | 3 | 25% |
| namespaces, XML schemas, document object model, presenting XML | | 3 | 3 | |
| using CSS, XSLT, XPath, XQuery, FLOWR | | 3 | 3 | |
| Features, JSON vs. XML, JSON Data Types, JSON Objects, JSON | | 3 | 3 | |
| Arrays, JSON HTML. | | 3 | 1 | |

## 1. Define XML and elaborate on its syntax with an example.

XML (eXtensible Markup Language) is used to describe, store, and transport data. It is **both human-readable and machine-readable**, and its syntax is strict to ensure data consistency.

**Key Syntax Rules of XML:**

1. **XML Declaration (Optional but Recommended)**
   - Appears at the top of the XML file.
   - Specifies version and encoding.

```
<?xml version="1.0" encoding="UTF-8"?>
```

2. **Root Element**
   - Every XML document must have **one and only one root element** that encloses all other elements.

```
<bookstore>
    ...
</bookstore>
```

3. **Well-Formed Tags**
   - Every tag must have a **matching closing tag**.
   - Tags are **case-sensitive** (<Book> ≠ <book>).

```
<book>XML Basics</book>
```

4. **Nesting Must Be Proper**
   - Elements must be **properly nested**, not overlapping.

```
<book>
    <title>XML Guide</title>
</book>
```

```
<book>
    <title>XML Guide</book></title>
```

5. **Attributes Must Be Quoted**
   - Attribute values should always be enclosed in **single or double quotes**.

```
<book id="101" category="programming">
    <title>Learn XML</title>
</book>
```

6. **Element Content**
   - Elements can contain **text, other elements**, or both.

```
<author>
    <name>ABC</name>
    <age>30</age>
</author>
```

7. **Comments in XML**

- Comments can be added like this:

```
<!-- This is a comment -->
```

8. **Empty Elements**
   - Can be written using self-closing tags.

```
<line-break />
```

**Sample XML Document with Syntax:**

```
<?xml version="1.0" encoding="UTF-8"?>
<student id="101">
    <name>Ravi Kumar</name>
    <course>Computer Science</course>
    <age>21</age>
</student>
```

## 2. Explain the XML document structure with an example.

An XML (eXtensible Markup Language) document is a structured text format used to store and transport data. It is both human-readable and machine-readable.

**Structure of an XML Document includes:**

1. **XML Declaration (optional but recommended)**
   - Appears at the top of the XML file.
   - Specifies version and encoding.

```
<?xml version="1.0" encoding="UTF-8"?>
```

2. **Root Element (mandatory)**
   - Every XML document must have one and only one root element.

3. **Child Elements**
   - Elements nested inside the root element.
4. **Attributes**
   - Provide additional information about elements.
5. **Comments (optional)**
   - Comments are used to describe or annotate parts of the XML.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
    <book category="Fiction">
        <title lang="en">The Alchemist</title>
        <author>Paulo Coelho</author>
        <year>1988</year>
        <price>9.99</price>
    </book>
</bookstore>
```

**3. What is a Document Type Definition (DTD)? Write a DTD for an XML document that represents a book with title, author, year and price.**

**Definition of Document Type Definition (DTD)**

A **Document Type Definition (DTD)** is a set of rules that defines the structure and the legal elements and attributes of an XML document. It specifies which elements are allowed in the document, their relationships, and the data types that they can contain.

```xml
<book>
  <title>The Alchemist</title>
  <author>Paulo Coelho</author>
  <year>1988</year>
  <price>399.00</price>
</book>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book [
  <!ELEMENT book (title, author, year, price)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT author (#PCDATA)>
  <!ELEMENT year (#PCDATA)>
  <!ELEMENT price (#PCDATA)>
]>
```

**4. Create a DTD that defines the structure for storing student information.**

```
<students>
    <student>
        <name>Nikhil</name>
        <rollno>101</rollno>
        <department>Computer Science</department>
        <email>nikhil@example.com</email>
    </student>
</students>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE students [
    <!ELEMENT students (student+)>
    <!ELEMENT student (name, rollno, department, email)>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT rollno (#PCDATA)>
    <!ELEMENT department (#PCDATA)>
    <!ELEMENT email (#PCDATA)>
]>
```

- <!ELEMENT students (student+)>

  The root element <students> contains **one or more** <student> elements.

- <!ELEMENT student (name, rollno, department, email)>

  Each <student> contains **four sub-elements** in order: name, rollno, department, email.

- <!ELEMENT name (#PCDATA)>

  The element contains parsed character data (text).

**6. Create a DTD for the following XML document.**

```
<car-info>
    <car>
        <model>ABC</ model >
        <color>red</color>
        <year>2019</year>
    </car>
</car-info>
```

```
<!ELEMENT car-info (car)>
<!ELEMENT car (model, color, year)>
<!ELEMENT model (#PCDATA)>
<!ELEMENT color (#PCDATA)>
<!ELEMENT year (#PCDATA)>
```

**7. Create a DTD for an XML document representing a student with name, roll number, department, and email.**

```
<?xml version="1.0"?>
<!DOCTYPE student [
  <!ELEMENT student (name, rollno, department, email)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT rollno (#PCDATA)>
  <!ELEMENT department (#PCDATA)>
  <!ELEMENT email (#PCDATA)>
]>
<student>
  <name>Ravi Kumar</name>
  <rollno>CS102</rollno>
  <department>Computer Science</department>
  <email>ravi.kumar@example.com</email>
</student>
```

**8. Explain the structure of DTD with an example for a product containing product name, ID, price, and quantity.**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE product [
  <!ELEMENT product (productName, productID, price, quantity)>
  <!ELEMENT productName (#PCDATA)>
  <!ELEMENT productID (#PCDATA)>
  <!ELEMENT price (#PCDATA)>
  <!ELEMENT quantity (#PCDATA)>
]>
<product>
  <productName>Smartphone</productName>
  <productID>SP1001</productID>
  <price>15999.00</price>
  <quantity>250</quantity>
</product>
```

**9. Elaborate on the structure and components of XML Schema Definition (XSD), and illustrate with an example.**

Structure and Components of XSD:

- **Root Element (**<xs:schema>**)**

The schema begins with <xs:schema>, which defines the XML namespace for schema elements.

- **Element Declaration (**<xs:element>**)**

  Used to define elements that can appear in the XML document.

- **Complex Types (**<xs:complexType>**)**

  Defines elements that contain other elements or attributes.

- **Simple Types (**<xs:simpleType>**)**

  Used to define elements or attributes that contain only text (like string, date, integer, etc.).

- **Sequence (**<xs:sequence>**)**

  Specifies that the child elements must appear in a specific order.

- **Data Types**

  Built-in types like xs:string, xs:integer, xs:date, xs:gYear, etc.

- **Attributes (**<xs:attribute>**)**

  Optional: defines additional information within an element.

```
<student>
    <name>John</name>
    <age>20</age>
</student>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="student">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="age" type="xs:integer"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

**10.Define XSD. Create a XML schema for the following XML documents.**

```xml
<car-info>
    <car>
        <model>ABC</ model >
        <color>red</color>
        <year>2019</year>
    </car>
</car-info>
```

An XML Schema Definition (XSD) is a standard for describing the structure and data types of an XML document. It provides a way to define the elements, attributes, and data types that are allowed in an XML document.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="car-info">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="car">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="model" type="xs:string"/>
              <xs:element name="color" type="xs:string"/>
              <xs:element name="year" type="xs:gYear"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

**11. Design an XML Schema Definition (XSD) to validate an XML document containing student information, including fields such as name, roll number, department, email, and age.**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="student">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="roll" type="xs:string"/>
        <xs:element name="department" type="xs:string"/>
        <xs:element name="email" type="xs:string"/>
        <xs:element name="age" type="xs:integer"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:string" use="optional"/>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

## 12. What is XSLT? Explain how it is used to transform XML documents with a suitable example.

**What is XSLT?**

XSLT (eXtensible Stylesheet Language Transformations) is a language used to transform XML documents into other formats such as HTML, plain text, or another XML document.

It is part of the XSL (Extensible Stylesheet Language) family and works using stylesheets that describe how to process and present the XML data.

**How XSLT is used to transform XML:**

- An **XSLT stylesheet** is written to define how elements in the XML should be transformed.
- An **XSLT processor** reads both the XML and the XSLT, and outputs the transformed result.
- You match XML elements with <xsl:template> and apply transformation logic.

**XML File – students.xml**

```xml
<students>
   <student>
      <name>Ravi</name>
      <grade>A</grade>
   </student>
   <student>
      <name>Priya</name>
      <grade>B</grade>
   </student>
</students>
```

**XSLT Stylesheet – students.xsl**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <body>
        <h2>Student Grades</h2>
        <table border="1">
          <tr>
            <th>Name</th>
            <th>Grade</th>
          </tr>
          <xsl:for-each select="students/student">
            <tr>
              <td><xsl:value-of select="name"/></td>
              <td><xsl:value-of select="grade"/></td>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

**Result (HTML output after transformation):**

```
<html>
  <body>
    <h2>Student Grades</h2>
    <table border="1">
      <tr><th>Name</th><th>Grade</th></tr>
      <tr><td>Ravi</td><td>A</td></tr>
      <tr><td>Priya</td><td>B</td></tr>
    </table>
  </body>
</html>
```

**13.Explain the XPath used in XML documents with an example.**

XPath (XML Path Language) is used to navigate through elements and attributes in an XML document. It allows you to locate specific parts of an XML file based on pat

```
<library>
    <book category="fiction">
        <title>Harry Potter</title>
        <author>J.K. Rowling</author>
        <price>29.99</price>
    </book>
    <book category="non-fiction">
        <title>A Brief History of Time</title>
        <author>Stephen Hawking</author>
        <price>15.00</price>
    </book>

 <book category="web">
        <title>world wide web</title>
        <author>R sabestha</author>
        <price>20.00</price>
    </book>
</library>
```

Here’s an overview of how XPath works in XML:

1. **Selects the root element.** Root Node (**/**)**:**

   /rootElement

2. **Selects nodes anywhere in the document from the current node**.

   //elementName

3. **Direct Child (/)**: Selects a child element.

   /library/book

4. **All Descendants (//)**: Selects all elements matching the name, regardless of depth.

   //author

5. **Attributes (@)**: Selects an attribute of an element.

   //book[@category="fiction"]

6. **By Index**: Selects a specific instance of an element.

   /library/book[1]  <!-- Selects the first book element -->

7. **By Condition**: Selects nodes based on attribute values or text.

   //book[@category='fiction'] <!-- Selects book with category attribute as 'fiction' -->

   //book[price > 20] <!-- Selects books where the price is greater than 20 -->

8. Functions in XPath

9. **text()**: Selects the text content of a node.

   //book/title/text()

1. **contains()**: Tests if a node contains a specified text.

   //book[contains(title, 'XML')]

10. **last()**: Selects the last node in a list.

    /library/book[last()]

## Explanation of XPath Syntax in These Examples

- `/`: Starts from the root.
- `@`: Selects attributes.
- `[ ]`: Filters nodes by conditions (like index or attribute value).
- `text()`: Gets text content within an element.
- `contains()`: Checks if a node contains specified text.
- `last()`: Refers to the last item in a selection.

# XQuery

**XQuery** (XML Query Language) is a powerful language for querying and manipulating XML data. It is particularly useful for extracting and transforming data from XML documents, databases, and web services. XQuery shares many functions and expressions with XPath but adds greater functionality for complex querying, data manipulation, and conditional operations.

XQuery expressions are often written in FLWOR (For, Let, Where, Order by, Return) expressions to iterate, filter, and return results.

## Explanation of FLWOR Syntax

- `for`: Iterates over nodes.
- `let`: Binds values to variables.
- `where`: Filters nodes based on a condition.
- `order by`: Sorts the result.
- `return`: Specifies what should be returned for each item.

## Example XML Document

We'll use the same XML structure as before:

```
<library>
<book id="1" category="fiction">
    <title>The Great Gatsby</title>
    <author>F. Scott Fitzgerald</author>
    <price>10.99</price>
</book>
<book id="2" category="fiction">
    <title>To Kill a Mockingbird</title>
    <author>Harper Lee</author>
    <price>7.99</price>
</book>
<book id="3" category="non-fiction">
    <title>Brief Answers to the Big Questions</title>
    <author>Stephen Hawking</author>
    <price>15.99</price>
</book>
</library>
```

# XQuery Expressions and Examples

## 1. Simple Query to Retrieve All Titles

```
for $book in /library/book return $book/title
```

### Result:

- The Great Gatsby
- To Kill a Mockingbird
- Brief Answers to the Big Questions

## 2. Using `where` to Filter Books by Category

```
for $book in /library/book
where $book/@category = 'fiction' return $book/title
```

### Result:

- The Great Gatsby
- To Kill a Mockingbird

## 3. Filtering and Sorting Books by Price

```
for $book in /library/book where
    $book/price > 10
order by $book/price ascending return
    <book>
            <title>{ $book/title }</title>
            <price>{ $book/price }</price>
        </book>
```

### Result:

```
<book>
    <title>The Great Gatsby</title>
    <price>10.99</price>
</book>
<book>
    <title>Brief Answers to the Big Questions</title>
    <price>15.99</price>
</book>
```

## 4. Using `let` to Bind Variables

- Suppose you want to format the result with both title and author:

```
for $book in /library/book let $title :=
    $book/title let $author :=
    $book/author
return <bookInfo>{ $title } by { $author }</bookInfo>
```

### Result:

```
<bookInfo>The Great Gatsby by F. Scott Fitzgerald</bookInfo>
<bookInfo>To Kill a Mockingbird by Harper Lee</bookInfo>
<bookInfo>Brief Answers to the Big Questions by Stephen Hawking</bookInfo>
```

## 5. Aggregating Data (e.g., Total Price of All Books)

```
let $total := sum(/library/book/price) return <totalPrice>{
    $total }</totalPrice>
```

**Result**:

<totalPrice>34.97</totalPrice>

## 6. Creating New XML Structures with XQuery

- XQuery can create new XML documents by transforming existing XML data:

```
for $book in /library/book return
    <bookSummary>
            <title>{ $book/title }</title>
            <category>{ $book/@category }</category>
        </bookSummary>
```

Result:

```
<bookSummary>
    <title>The Great Gatsby</title>
    <category>fiction</category>
</bookSummary>
<bookSummary>
    <title>To Kill a Mockingbird</title>
    <category>fiction</category>
</bookSummary>
<bookSummary>
    <title>Brief Answers to the Big Questions</title>
    <category>non-fiction</category>
</bookSummary>
```

**DOM in XML**

**1. What is DOM?**

**DOM** stands for **Document Object Model**.

It is a programming interface for XML (and HTML) documents that represents the document as a **tree structure** of nodes, where each node is an object representing a part of the document (elements, attributes, text, etc.).

### 3. Structure of DOM:

The XML document is converted into a **tree**:

- **Document Node** → Root of the XML
- **Element Nodes** → Tags in XML
- **Attribute Nodes** → Tag attributes
- **Text Nodes** → Text inside tags

```
<bookstore>
    <book category="fiction">
        <title>Harry Potter</title>
        <author>J.K. Rowling</author>
        <year>1997</year>
    </book>
</bookstore>
```

DOM Tree Representation:

```
Document
 └── Element: bookstore
      └── Element: book (attribute: category="fiction")
           ├── Element: title → Text: Harry Potter
           ├── Element: author → Text: J.K. Ro↓g
           └── Element: year → Text: 1997
```

# JSON

**JSON** (JavaScript Object Notation) is a lightweight, text-based data interchange format. It is designed to be simple, easy to read and write, and language-independent, although it is based on JavaScript syntax.

JSON is widely used for transferring data between a server and a web application or between systems due to its simplicity and ease of parsing.

### Features of JSON

1. **Simplicity**: JSON is lightweight and easier to read and write compared to XML.
2. **Data Format**: It represents data as key-value pairs, arrays, and nested objects.
3. **Readability**: JSON's syntax is minimal, resembling plain JavaScript, which makes it more human-readable.
4. **Data Types**: JSON supports numbers, strings, arrays, booleans, null, and objects.
5. **Usage**: Primarily used in web APIs, JavaScript-based applications, and modern web technologies.
6. **Compatibility**: JSON works seamlessly with JavaScript and is supported by most modern programming languages.
7. **Parsing**: JSON parsers are typically faster and use less memory than XML parsers.
8. **Schema**: JSON Schema is available for defining and validating JSON structure but is less formal than XML schemas (e.g., XSD).

### Comparison: JSON vs. XML

| Feature | JSON | XML |
|---|---|---|
| **Simplicity** | Simpler syntax, easier to read/write | More verbose due to opening/closing tags |

| Feature | JSON | XML |
|---|---|---|
| **Data Type Support** | Numbers, strings, arrays, Booleans, null | Everything is text unless parsed |
| **Schema Validation** | JSON Schema | XSD, DTD |
| **Readability** | More human-readable | less readable |
| **Namespaces** | No direct support | Fully supported |
| **Metadata** | Metadata stored as part of the structure | Attributes provide better metadata support |
| **Performance** | Faster to parse | Slower parsing due to verbosity |

**The data types in JSON:**

---

**1. String**

- Represents textual data.
- Enclosed in double quotes (").
- Can include escape characters (e.g., \n, \t, \").
- Example:

  "name": "Alice"

---

**2. Number**

- Represents numeric values, including:
  - Integers: 1, 100, -42
  - Floating-point numbers: 3.14, -0.001
- No distinction between integers and floats; all are treated as Number.
- No support for special types like NaN or Infinity.
- Example:

  "price": 19.99

### 3. Boolean

- Represents logical values.
- Only two possible values: true or false (without quotes).
- Example:

  "isAvailable": true

---

### 4. Null

- Represents the absence of a value.
- Always written as null (without quotes).
- Example:

  "middleName": null

---

### 5. Array

- Represents an ordered list of values.
- Enclosed in square brackets ([]).
- Can contain multiple types, including nested arrays or objects.
- Example:

  "colors": ["red", "green", "blue"]

---

### 6. Object

- Represents an unordered collection of key-value pairs.
- Enclosed in curly braces ({}).
- Keys are always strings, and values can be any JSON data type.
- Example:

```
"person": {
  "name": "Alice",
  "age": 25,
  "isStudent": false
}
```

---

### 7. Composite Structures

- JSON data types can be nested, allowing for complex and hierarchical structures.
- Example combining all data types:

```
{
  "id": 101,
  "name": "John Doe",
  "isEmployee": true,
  "address": null,
  "skills": ["JavaScript", "Python", "SQL"],
  "profile": {
    "department": "IT",
    "yearsOfExperience": 5
  }
}
```

---

**JSON Data Types**

| Data Type | Description | Example |
|---|---|---|
| String | Textual data in double quotes | "Hello, World!" |
| Number | Integer or floating-point numbers | 42, 3.14 |
| Boolean | Logical true or false values | true, false |
| Null | Represents no value | null |
| Array | Ordered list of values | ["apple", "banana"] |
| Object | Unordered collection of key-value pairs | {"key": "value"} |

**The syntax of JSON**

---

**Basic Rules**

1. **Objects** are enclosed in curly braces {}.
2. **Arrays** are enclosed in square brackets [].
3. **Key-Value Pairs**:

- Keys (field names) are always strings and must be enclosed in double quotes (").
- Values can be strings, numbers, objects, arrays, booleans, or null.

4. **Separators**:
   - Key-value pairs are separated by colons (:).
   - Multiple key-value pairs or array elements are separated by commas (,).
5. **Whitespace**: Ignored, allowing for formatting flexibility (e.g., spaces, tabs, and newlines).

## What is a JSON Object?

A **JSON Object** is a structured representation of data enclosed in curly braces {}. It consists of one or more key-value pairs, where keys are strings and values can be any valid JSON data type. JSON Objects allow you to organize and represent hierarchical data.

---

### Syntax of a JSON Object

1. **Curly Braces**: JSON objects are enclosed in {}.
2. **Key-Value Pairs**:
   - Keys must be strings enclosed in double quotes (").
   - Values can be of any JSON data type (string, number, object, array, boolean, or null).
3. **Separators**:
   - A colon (:) separates each key and its value.
   - Commas (,) separate multiple key-value pairs.

---

### Example of a JSON Object

```
{
  "name": "Alice",
  "age": 30,
  "isStudent": false,
  "skills": ["JavaScript", "Python", "SQL"],
  "address": {
    "street": "123 Main St",
    "city": "New York",
    "zipcode": "10001"
  }
}
```

## JSON Arrays

A **JSON array** is an ordered list of values, similar to arrays in most programming languages. It is enclosed in square brackets [] and can hold multiple data types, including objects, other arrays, or primitive values like strings, numbers, Booleans, and null.

---

### 1. Syntax of a JSON Array

```
[
  "value1",
  "value2",
  "value3"
]
```

**Example**:

```
{
  "fruits": ["apple", "banana", "cherry"]
}
```

---

### 2. Features of JSON Arrays

- Values inside the array are **comma-separated**.
- Can include **different data types** in the same array.
- Can be nested to include arrays or objects as elements.

**Example with Mixed Data Types**:

```
{
   "mixedArray": ["text", 123, true, null, {"key": "value"}]
}
```

**Nested Arrays**:

```
{
   "nestedArray": [[1, 2], [3, 4], [5, 6]]
}
```

---

## 3. Accessing JSON Arrays

When working with JSON arrays, the way you access their values depends on the programming language you're using.

*JavaScript Example:*

```
// JSON array
const jsonArray = ["apple", "banana", "cherry"];

// Access by index
Document.write(jsonArray[0]); // Output: "apple"



// Iterate through the array
jsonArray.forEach(item => document.write(item));
// Output: "apple", "banana", "cherry"
```

## 4. Manipulating JSON Arrays

*Adding Elements*

- **JavaScript**:

```
const fruits = ["apple", "banana"];
fruits.push("cherry"); // Add an element
console.log(fruits);   // Output: ["apple", "banana", "cherry"]
```

*Removing Elements*

- **JavaScript**:

```
const fruits = ["apple", "banana", "cherry"];
fruits.pop(); // Removes the last element
console.log(fruits);   // Output: ["apple", "banana"]
```

*Filtering Elements*

- **JavaScript**:

```
const numbers = [1, 2, 3, 4, 5];
const even = numbers.filter(num => num % 2 === 0);
console.log(even); // Output: [2, 4]
```

---

## 5. JSON Arrays with Objects

A common use case of JSON arrays is to store a collection of objects.

**Example**:

```
 [
   {"name": "Alice", "age": 25},
   {"name": "Bob", "age": 30},
   {"name": "Charlie", "age": 35}
]
```

*JavaScript Example:*

javascript
Copy code

```
const users = [
   { name: "Alice", age: 25 },
   { name: "Bob", age: 30 },
   { name: "Charlie", age: 35 }
];

// Access object properties
console.log(users[0].name); // Output: "Alice"

// Iterate through objects
users.forEach(user => console.log(user.name));
// Output: "Alice", "Bob", "Charlie"
```

---

# JSON and HTML: An Overview

JSON (JavaScript Object Notation) and HTML (HyperText Markup Language) often work together in web development. JSON is used for data exchange, while HTML structures and presents content in the browser. Here's how they interact:

---

## 1. Embedding JSON in HTML

You can embed JSON data in an HTML document to share or dynamically use it in your scripts.

*Example: JSON in an HTML <script> Tag*

```
<!DOCTYPE html>
<html>
<head>
   <title>JSON Example</title>
</head>
<body>
   <h1>JSON in HTML</h1>

   <script id="json-data" type="application/json">
      {
         "name": "Kumara",
         "age": 25,
         "hobbies": ["reading", "cycling", "gaming"],
          "marks": [12,34,26]
      }
   </script>

   <script>
      // Access JSON data from the script tag
      const jsonData = JSON.parse(document.getElementById("json-
data").textContent);
      document.writeln("Name"+jsonData.name);
  </script>
</body>
</html>
```