WEB TECHNOLOGY
Unit – IV

Contents

Enterprise JavaBean

Creating a JavaBean

A **JavaBean** is like a blueprint for creating small building blocks in Java programming. These blocks can store and manage data, and they can be reused in different programs. Here's a simple way to understand how to create a JavaBean step by step:

1. **Create a Java Class**

A JavaBean starts as a normal Java class. Think of a class as a container where you define the data and actions you want to include.

Example:

```
public class Student {
}
```

2. **Make the Class Serializable**

Serialization allows the JavaBean to be saved to a file or sent over the internet. To make a class serializable, you add implements Serializable.

Example:

```
import java.io.Serializable;

public class Student implements Serializable {
}
```

3. **Define Private Variables (Fields)**

These variables store the data for the JavaBean. They are kept private to ensure the data can only be accessed through specific methods.

Example:

```
import java.io.Serializable;

public class Student implements Serializable {
    private String name; // To store the name
    private int rollNumber; // To store the roll number
}
```

4. **Add Public Getter and Setter Methods**

Getter and setter methods let you access and modify the private variables safely. They follow a standard naming rule:

Getter: Starts with get followed by the variable name in camel case.

Setter: Starts with set followed by the variable name in camel case.

Example:

```
public String getName() { // Getter for 'name'
    return name;
}

public void setName(String name) { // Setter for 'name'
    this.name = name;
}
```

```
public int getRollNumber() { // Getter for 'rollNumber'
    return rollNumber;
}

public void setRollNumber(int rollNumber) { // Setter for 'rollNumber'
    this.rollNumber = rollNumber;
}
```

5. **Final JavaBean Example**

Here is what the complete JavaBean for a Student would look like:

```
import java.io.Serializable;

public class Student implements Serializable {
    private String name; // Private field
    private int rollNumber; // Private field

    // Getter for 'name'
    public String getName() {
        return name;
    }

    // Setter for 'name'
    public void setName(String name) {
        this.name = name;
    }

    // Getter for 'rollNumber'
    public int getRollNumber() {
        return rollNumber;
```

```
    }

    // Setter for 'rollNumber'
    public void setRollNumber(int rollNumber) {
        this.rollNumber = rollNumber;
    }
}
```

Key Points to Remember

1. **Serializable**: Add implements Serializable to the class to save or transfer the object.

2. **Private Fields**: Keep variables private for security.

3. **Getter and Setter Methods**: Use these methods to read and modify the values of variables.

4. **Naming Rules**: Follow the proper naming conventions for getter and setter methods.

JavaBeans Properties

JavaBeans Properties are the pieces of data that a JavaBean stores. They represent the characteristics or attributes of the JavaBean. For example, if a JavaBean represents a Student, its properties could be the name, roll number, or grade of the student.

Here's a step-by-step explanation of JavaBeans properties:

What Are Properties?

- Properties are variables inside the JavaBean that hold data.

- They are declared as private, meaning they cannot be accessed directly from outside the JavaBean.

- To access or change the value of these properties, we use getter and setter methods.

Types of Properties

There are three main types of properties in a JavaBean:

1. **Readable Property**: You can only read its value using a getter method.

Example: public int getRollNumber()

2. **Writable Property**: You can only change its value using a setter method.

Example: public void setRollNumber(int rollNumber)

3. **Read-Write Property**: You can both read and modify the value using getter and setter methods.

Defining Properties in a JavaBean

To define a property:

1. Declare a private variable to store the data.

2. Create a getter method to read the property.

3. Create a setter method to change the property.

Example of JavaBeans Properties

Let's create a Student JavaBean with two properties: name and rollNumber.

```
import java.io.Serializable;

// The JavaBean class
```

```
public class Student implements Serializable {
    // Properties (private variables)
    private String name;
    private int rollNumber;

    // Getter method for 'name' (Readable property)
    public String getName() {
        return name;
    }

    // Setter method for 'name' (Writable property)
    public void setName(String name) {
        this.name = name;
    }

    // Getter method for 'rollNumber' (Readable property)
    public int getRollNumber() {
        return rollNumber;
    }

    // Setter method for 'rollNumber' (Writable property)
    public void setRollNumber(int rollNumber) {
        this.rollNumber = rollNumber;
    }
}
```

Accessing Properties

To use the properties, you first create an object of the Student JavaBean and then call the getter and setter methods.

Example Usage:

```
public class Main {
    public static void main(String[] args) {
        Student student = new Student();

        // Setting properties using setter methods
        student.setName("John");
        student.setRollNumber(101);

        // Accessing properties using getter methods
        System.out.println("Name: " + student.getName()); // Output: Name: John
        System.out.println("Roll Number: " + student.getRollNumber()); // Output: Roll Number: 101
    }
}
```

Key Rules for JavaBeans Properties

1. **Private Variables**: Properties are always private.

2. **Getter Methods**: Used to read the value of the property.

Format: public DataType getPropertyName()

3. **Setter Methods**: Used to modify the value of the property.

Format: public void setPropertyName(DataType value)

4. **Naming Convention**: Property names follow camelCase (e.g., studentName, rollNumber).

Why Are Properties Important?

- They ensure encapsulation: Data is hidden and accessed securely.

- They make the JavaBean reusable and easier to integrate into different applications.

- They provide a clear and organized way to manage data in programs.

Enterprise Java Bean (EJB) Types

In Enterprise Java Beans (EJB), there are different types of beans used to manage different tasks in an application. These beans help in handling logic, storing data, and managing user interactions. The main types of beans in EJB are:

1. **Stateful Session Bean**

Definition: A Stateful Session Bean keeps information (state) about a single client across multiple interactions.

Example: Think of it as a shopping cart on an e-commerce site. The cart remembers the items you add until you checkout.

Key Features:

- Each client gets its own instance of the bean.

- The bean maintains its state (data) between different method calls.

- When the client session ends, the state is discarded.

Example Use Case:

- Managing shopping carts.

- Storing temporary user information during a login session.

2. **Stateless Session Bean**

Definition: A Stateless Session Bean does not keep any information (state) between method calls. It is used when each request is independent.

Example: Think of it like a calculator. You input numbers, get the result, and nothing is remembered for the next calculation.

Key Features:

- The bean doesn’t store client-specific data.

- It can handle multiple requests from different clients because it doesn’t maintain any state.

- It is lightweight and efficient.

Example Use Case:

- Performing simple operations like calculations or data validation.

- Handling independent tasks like sending emails or processing payments.

3. **Entity Bean**

Definition: An Entity Bean represents data stored in a database. It acts as a bridge between the Java program and the database.

Example: Think of it like a student's record stored in a database. Each record (student) has properties like name, roll number, and grade, which are mapped to the bean.

Key Features:

- It is used to manage and store data persistently in a database.

- Each Entity Bean instance is associated with a specific database record.

- It automatically updates the database when the bean is modified.

Example Use Case:

- Storing user profiles in a database.

- Managing inventory or employee data.

Summary

- **Stateful Session Bean**: Keeps information specific to a user.

- Stateless Session Bean: Does not keep any information, used for simple tasks.

- **Entity Bean**: Manages data stored in a database.

These beans make Java applications powerful by handling tasks efficiently and securely. Let me know if you need further clarification!

Node.js: Introduction

Definition:

Node.js is a tool that allows you to run JavaScript code outside of a web browser. Normally, JavaScript is used to make web pages interactive, but with Node.js, you can use JavaScript to build applications that run on servers.

Purpose:

Node.js is mainly used to build web applications that handle multiple user requests quickly. For example, when you send a message on a chat app, Node.js helps process that message almost instantly.

Features:

1. **Fast**: It is fast because it is built on a powerful engine called V8, created by Google.

2. **Non-blocking**: Node.js can handle many tasks at the same time without waiting for one task to finish before starting another.

3. **Open-source**: Anyone can use it for free.

4. **Cross-platform**: It works on Windows, macOS, and Linux.

Environment Setup

To start using Node.js, you need to set up your computer with the necessary tools. Follow these steps:

Step 1: Download and Install Node.js

1. Go to the official Node.js website: https://nodejs.org.

2. Choose the version suitable for your computer (LTS is recommended for beginners).

3. Download the installer and follow the instructions to install it.

4. During installation, Node.js will also install a tool called npm (Node Package Manager), which helps you install other tools and libraries.

Step 2: Verify Installation

1. Open your computer's terminal or command prompt.

2. Type node -v and press Enter. If it shows a version number, Node.js is successfully installed.

3. Type npm -v to check if npm is installed.

Step 3: Write Your First Node.js Program

1. Open a text editor like Notepad or Visual Studio Code.

2. Write the following code:

```
console.log("Hello, Node.js!");
```

3. Save the file as app.js.

4. Open the terminal, navigate to the folder where you saved the file, and type:

```
node app.js
```

5. You should see Hello, Node.js! printed on the screen.

Why Learn Node.js?

- It is used by big companies like Netflix, LinkedIn, and PayPal to build fast and reliable apps.

- It's a great choice if you want to become a full-stack developer (someone who works on both the front-end and back-end of web apps).

REPL Terminal

Definition:

REPL stands for Read-Eval-Print Loop. It is a tool that lets you write and test small pieces of code directly in the terminal.

What it Does:

- Read: It takes the code you type.

- Eval: It evaluates (runs) the code.

- Print: It shows the result of the code.

- Loop: It keeps repeating this process until you quit.

How to Use REPL:

1. Open your terminal or command prompt.

2. Type node and press Enter to start the REPL.

3. You can now type JavaScript commands like:

> console.log("Hello, REPL!");

Hello, REPL!

4. To exit REPL, type .exit or press Ctrl + C twice.

Why It's Useful:

It's great for quickly testing code without creating files or running full programs.

NPM (Node Package Manager)

Definition:

NPM is a tool that comes with Node.js. It helps developers easily share, download, and manage code libraries or packages.

What are Packages?

Packages are pre-written pieces of code that solve specific problems, like sending emails, connecting to a database, or building a web server.

How NPM Works:

1. To install a package, use the command:

npm install <package-name>

Example: To install a package called express:

npm install express

2. Once installed, you can use the package in your project.

Why NPM is Important:

- Saves time by reusing code.

- Helps manage all the libraries your project needs in one place.

- Offers access to thousands of free, open-source packages.

Callbacks Concept

Definition:

A callback is a function that you pass to another function to run after some task is finished.

Why Use Callbacks?

In Node.js, tasks like reading files or fetching data from a server take time. Instead of waiting for one task to finish, Node.js uses callbacks to continue doing other tasks in the meantime. This makes it non-blocking and fast.

Example:

Imagine ordering a pizza. Instead of standing at the counter waiting for your pizza, you give your phone number (callback) to the shop. They call you (callback is executed) when the pizza is ready.

Summary:

REPL Terminal: A tool to test small pieces of code directly in the terminal.

NPM: A tool to install and manage code libraries (packages) for Node.js projects.

Callbacks: A way to handle tasks that take time by running another function once the task is complete.

Events in Node.js

Definition:

In Node.js, an event is an action or occurrence (like a mouse click, a file being read, or a timer completing) that the program can listen for and respond to.

How Events Work:

Node.js has an event-driven architecture, which means it listens for events and runs specific code (called event handlers) when those events happen.

Example:

Imagine you're at a train station. When a train arrives (event), an announcement (event handler) is made. Similarly, in Node.js, events trigger specific responses.

EventEmitter:

Node.js has a built-in module called events that allows you to create, listen for, and handle events.

Example in code:

```
const EventEmitter = require('events');
```

```
const event = new EventEmitter();

// Create an event listener
event.on('greet', function() {
    console.log('Hello! Someone triggered the greet event.');
});

// Trigger the event
event.emit('greet');
```

Output: Hello! Someone triggered the greet event.

Why Events are Important:

Events allow Node.js to handle multiple tasks efficiently without waiting for each task to finish. For example, when a server receives multiple requests, it can handle them all using events.

Packaging in Node.js

Definition:

Packaging in Node.js refers to bundling your project's files and dependencies (libraries it uses) into a structure that others can easily use or download.

What is a Package?

A package is a folder or archive that contains code, configuration files, and other resources needed for a project.

It can be shared with others using tools like npm (Node Package Manager).

Creating a Package:

To create a package in Node.js, you use a file called package.json. This file contains important information about your project, such as:

The project's name, version, and description.

The dependencies (libraries) it uses.

Steps to Create a Package:

1. **Initialize the Project**:

Run the following command in the terminal:

npm init

It will ask questions about your project (like name, version, and description) and create a package.json file.

2. **Add Dependencies**:

To install a library, use:

npm install <library-name>

Example:

npm install express

This will add express as a dependency in package.json.

3. **Publish the Package (Optional)**:

If you want to share your package with others, you can publish it on the npm registry.

Why Packaging is Important:

- Makes it easy to share and reuse code.

- Keeps track of the libraries your project uses.

- Simplifies setting up and maintaining large projects.

Summary:

Events: Actions that Node.js can listen for and respond to, such as user actions or system triggers. They make applications fast and responsive.

Packaging: The process of organizing and managing a project's files and dependencies using tools like npm and package.json. It helps in sharing and maintaining code.

Express Framework

Definition:

Express is a lightweight and powerful web application framework built on top of Node.js. It simplifies the process of creating web applications and APIs (a way for software programs to communicate).

Why Use Express?

Without Express, creating a web application in Node.js can be complicated because you would have to write a lot of code to handle tasks like routing (deciding which page to show) or handling user input.

Key Features of Express:

1. **Routing**: Easily manage URLs like /home or /about to show different pages or data.

2. **Middleware**: Add reusable code that runs before sending a response (e.g., checking user authentication).

3. **Easy to Use:** Makes building web servers much simpler compared to plain Node.js.

How to Install and Use Express:

1. **Install Express using npm**:

npm install express

2. **Write a simple server using Express**:

```
const express = require('express');
const app = express();

// Define a route
app.get('/', (req, res) => {
    res.send('Hello, Express!');
});

// Start the server
app.listen(3000, () => {
    console.log('Server is running on port 3000');
});
```

What Happens Here?

app.get('/'): When someone visits the homepage (/), it responds with "Hello, Express!".

app.listen(3000): Starts the server on port 3000.

RESTful API

Definition:

A RESTful API (Representational State Transfer API) is a way for applications to communicate over the internet using specific rules. It's like a waiter in a restaurant: it takes requests (like ordering food) and gives back responses (like serving the food).

Why Use RESTful APIs?

They make it easy for different applications (e.g., a mobile app and a web app) to talk to the same backend server.

How RESTful APIs Work:

RESTful APIs use HTTP methods (the rules of the internet) to handle tasks:

1. **GET**: Retrieve data (e.g., get a list of users).

2. **POST**: Send new data (e.g., add a new user).

3. **PUT**: Update existing data (e.g., change user details).

4. **DELETE**: Remove data (e.g., delete a user).

Example of RESTful API in Express:

```
const express = require('express');
const app = express();

// Middleware to parse JSON
app.use(express.json());

// Sample data
let users = [
    { id: 1, name: 'John' },
    { id: 2, name: 'Jane' }
];
```

```
// GET: Retrieve all users
app.get('/users', (req, res) => {
    res.json(users);
});

// POST: Add a new user
app.post('/users', (req, res) => {
    const newUser = { id: users.length + 1, name: req.body.name };
    users.push(newUser);
    res.json(newUser);
});

// Start the server
app.listen(3000, () => {
    console.log('Server running on port 3000');
});
```

What Happens Here?

GET /users: Sends a list of all users.

POST /users: Adds a new user to the list.

Benefits of RESTful APIs:

1. Simple and easy to understand.

2. Widely used in web and mobile app development.

3. Compatible with almost any programming language.

Summary:

Express Framework: A tool that makes creating web servers and applications in Node.js faster and easier.

RESTful API: A system that allows applications to communicate using standard rules (HTTP methods like GET, POST, etc.).

MongoDB

What is MongoDB?

Definition:

MongoDB is a type of database that stores data in a flexible, JSON-like format called documents instead of traditional tables like in SQL databases.

It is useful when you need to handle large amounts of data, like for websites or mobile apps.

Why Use MongoDB with Node.js?

1. Both are fast and scalable.

2. MongoDB works well with JavaScript, which is used in Node.js.

3. Together, they allow developers to easily store, update, and retrieve data from a database.

Create a Database in MongoDB

A database is like a folder that holds collections (similar to tables in SQL).

Steps to Create a Database:

1. **Install MongoDB:**

Make sure MongoDB is installed and running on your computer. You can download it from the official MongoDB website.

2. **Install MongoDB Driver for Node.js:**

To connect MongoDB with Node.js, install the driver:

npm install mongodb

3. **Connect to MongoDB and Create a Database:**

Use the following code to create a database:

```
const { MongoClient } = require('mongodb');
const url = 'mongodb://localhost:27017'; // MongoDB server URL
const client = new MongoClient(url);

async function createDatabase() {
    try {
        await client.connect(); // Connect to the server
        console.log('Connected to MongoDB!');

        const db = client.db('myDatabase'); // Create or select a database
        console.log('Database created:', db.databaseName);
    } catch (err) {
        console.error('Error:', err);
    } finally {
        await client.close(); // Close the connection
    }
}

createDatabase();
```

What Happens Here?

- client.db('myDatabase'): Creates a database called myDatabase.

- If the database does not already exist, MongoDB creates it when you first add data.

Create a Collection in MongoDB

A collection is like a table in a database, where you store related data (documents). For example, a users collection might store user information.

Steps to Create a Collection:

1. **Connect to the Database and Create a Collection:**

Use the following code:

```
const { MongoClient } = require('mongodb');
const url = 'mongodb://localhost:27017';
const client = new MongoClient(url);

async function createCollection() {
    try {
        await client.connect();
        console.log('Connected to MongoDB!');

        const db = client.db('myDatabase'); // Connect to the database
        const collection = db.collection('myCollection'); // Create a collection

        console.log('Collection created:', collection.collectionName);
    } catch (err) {
        console.error('Error:', err);
    } finally {
        await client.close();
```

```
  }
}

createCollection();
```

What Happens Here?

- db.collection('myCollection'): Creates a collection named myCollection.

- If the collection does not already exist, MongoDB creates it when you first insert data.

Key Points to Remember:

1. MongoDB does not require you to define the structure of your data in advance. It's flexible and stores data as documents in collections.

2. A database groups related collections together.

3. A collection holds multiple documents (like rows in a table).

4. You need to install the MongoDB driver in Node.js to connect your application to MongoDB.

Simple Example:

Here's an example that combines both steps:

```
const { MongoClient } = require('mongodb');
const url = 'mongodb://localhost:27017';
const client = new MongoClient(url);

async function main() {
```

```
    try {
        await client.connect();
        console.log('Connected to MongoDB!');

        const db = client.db('school'); // Create a database called "school"
        const collection = db.collection('students'); // Create a collection called "students"

        console.log('Database and Collection created!');
    } catch (err) {
        console.error('Error:', err);
    } finally {
        await client.close();
    }
}

main();
```

Output:

- A database named school is created.

- A collection named students is created inside the school database.

Summary:

Create Database: Use client.db('<database-name>') to create or select a database.

Create Collection: Use db.collection('<collection-name>') to create a collection inside the database.

Insert

What is it?

The insert operation is used to add new data (documents) to a MongoDB collection.

Example Code:

```
const { MongoClient } = require('mongodb');
const url = 'mongodb://localhost:27017';
const client = new MongoClient(url);

async function insertData() {
    try {
        await client.connect();
        const db = client.db('school');
        const collection = db.collection('students');

        const newStudent = { name: 'John', age: 16, grade: '10th' };
        const result = await collection.insertOne(newStudent);

        console.log('Document inserted:', result.insertedId);
    } catch (err) {
        console.error('Error:', err);
    } finally {
        await client.close();
    }
}

insertData();
```

What Happens?

A new student document { name: 'John', age: 16, grade: '10th' } is added to the students collection.

Delete

What is it?

The delete operation removes data (documents) from a MongoDB collection.

Example Code:

```
async function deleteData() {
    try {
        await client.connect();
        const db = client.db('school');
        const collection = db.collection('students');

        const result = await collection.deleteOne({ name: 'John' });
        console.log('Documents deleted:', result.deletedCount);
    } catch (err) {
        console.error('Error:', err);
    } finally {
        await client.close();
    }
}

deleteData();
```

What Happens?

The document where name is John is removed from the students collection.

Update

What is it?

The update operation modifies existing data in a MongoDB collection.

Example Code:

```
async function updateData() {
```

```
    try {
        await client.connect();
        const db = client.db('school');
        const collection = db.collection('students');

        const result = await collection.updateOne(
            { name: 'John' }, // Find document
            { $set: { grade: '11th' } } // Update grade
        );

        console.log('Documents updated:', result.modifiedCount);
    } catch (err) {
        console.error('Error:', err);
    } finally {
        await client.close();
    }
}

updateData();
```

What Happens?

The grade of the document where name is John is updated to 11th.

Query

What is it?

A query is used to find data in a collection that matches specific conditions.

Example Code:

```
async function queryData() {
    try {
        await client.connect();
```

```
        const db = client.db('school');
        const collection = db.collection('students');

        const students = await collection.find({ grade: '10th' }).toArray();
        console.log('Students in 10th grade:', students);
    } catch (err) {
        console.error('Error:', err);
    } finally {
        await client.close();
    }
}

queryData();
```

What Happens?

Retrieves all documents where grade is 10th.

Sort

What is it?

The sort operation arranges documents in ascending or descending order based on a specific field.

Example Code:

```
async function sortData() {
    try {
        await client.connect();
        const db = client.db('school');
        const collection = db.collection('students');

        const sortedStudents = await collection.find().sort({ age: 1 }).toArray();
        console.log('Students sorted by age (ascending):', sortedStudents);
```

```
    } catch (err) {
        console.error('Error:', err);
    } finally {
        await client.close();
    }
}

sortData();
```

What Happens?

Documents are sorted in ascending order (1 for ascending, -1 for descending) by the age field.

Join

What is it?

A join operation combines data from multiple collections. In MongoDB, this is done using the $lookup aggregation stage.

Example Code:

```
async function joinData() {
    try {
        await client.connect();
        const db = client.db('school');
        const students = db.collection('students');
        const marks = db.collection('marks');

        const joinedData = await students.aggregate([
            {
                $lookup: {
                    from: 'marks', // Collection to join
                    localField: '_id', // Field in "students"
```

```
                foreignField: 'studentId', // Field in "marks"
                as: 'studentMarks' // Resulting array
            }
        }
    ]).toArray();

    console.log('Joined data:', joinedData);
  } catch (err) {
    console.error('Error:', err);
  } finally {
    await client.close();
  }
}

joinData();
```

What Happens?

Combines students collection with the marks collection based on matching _id in students and studentId in marks.

Summary of Operations:

1. **Insert**: Adds new data to a collection.

2. **Delete**: Removes data from a collection.

3. **Update**: Modifies existing data in a collection.

4. **Query**: Finds data that matches certain conditions.

5. **Sort**: Arranges data in a specific order (ascending/descending).

6. **Join**: Combines data from multiple collections.