MATS UNIVERSITY

NAAC GRADE A+ ACCREDITED UNIVERSITY

MATS CENTRE FOR DISTANCE & ONLINE EDUCATION

Advanced Java Programming

Master of Computer Applications (MCA)
Semester - 2

SELF LEARNING MATERIAL

Master of Computer Applications

ODL MCA DSC-201-T
Advanced Java Programming

Acknowledgement

COURSE INTRODUCTION

Java is a powerful, object-oriented programming language widely used for developing robust, scalable, and secure applications. This course provides a comprehensive understanding of object-oriented programming concepts, JavaFX for building graphical user int erfaces, and advanced web development technologies such as Servlets, JSP, and the Spring Framework. Students will gain both theoretical knowledge and hands-on experience in designing and developing modern Java applications.

Block 1: Object-Oriented Programming Concepts and Implementations

Object-oriented programming (OOP) enhances code reusability, scalability, and maint ainability. This Unit int roduces key OOP concepts such as encapsulation, inheritance, polymorphism, and abstraction. Students will learn how to implement OOP principles in Java, utilizing class es, objects, and design patterns for efficient software development.

Block 2: JavaFX Technology

JavaFX is a modern Java framework for developing rich graphical user int erfaces (GUIs). This Unit explores JavaFX components, event handling, layout management, and styling using CSS. Students will learn how to create int eractive desktop applications with advanced UI controls and multimedia int egration.

Block 3: Servlet Technology

Servlets are essential for developing dynamic web applications in Java. This Unit covers the fundamentals of Servlet technology, HTTP request/response handling, session management, and database connectivity using JDBC. Students will learn how to create server-side applications that handle web-based int eractions efficiently.

Block 4: JSP Technology

JavaServer Pages (JSP) enable the development of dynamic web pages by int egrating Java with HTML. This Unit int roduces JSP scripting elements, directives, custom tags, andexpression language (EL). Students will gain experience in

developing int eractive and data-driven web applications using JSP and Servlets.

Block 5: Spring and Spring Boot Framework

Spring is a powerful Java framework for building enterprise applications, while Spring Boot simplifies application development with pre-configured setups. This Unit explores Spring Core concepts, dependency injection, Spring MVC, and RESTful API development using Spring Boot. Students will learn how to build scalable and efficient Java applications using industry-standard frameworks.

Block 1: Object-Oriented Programming Concepts and Implementations

Unit 1: Object Oriented Programming Concepts and Implementation

Structure

1.1: Introduction

Object-Oriented Programming (OOP) is a fundamental programming paradigm, or way of thinking, that revolutionizes how we design and build software. Instead of focusing primarily on functions and logic (as in procedural programming), OOP centers around the concept of "objects." These objects are self-contained units that bundle together both data (attributes or properties) and the behaviors (methods or functions) that operate on that data. Think of a real-world object, like a car. It has data (color, make, model, current speed) and behaviors (accelerate, brake, turn). OOP allows us to model this car as a single "object" in our code, making the program structure more int uitive and closer to how we perceive the real world.

The foundation of OOP lies in the concept of a "class ," which acts as a blueprint for creating these objects. The class defines the common properties and methods that all objects of its type will have. This approach leads to highly modular and scalable software. By using core principles like encapsulation (hiding int ernal details), inheritance (building new   class es based on existing ones), and polymorphism (allowing objects to be treated in a uniform way, even if they behave differently), OOP helps developers create code that is more flexible,

reusable, and easier to maint ain and debug, especially for large, complex applications.

1.2: Learning Outcomes

Upon successful completion of this topic, you will be able to:

1. Define Object-Oriented Programming and contrast it with procedural programming.
2. Explain the four core principles of OOP: Encapsulation, Inheritance, Polymorphism, and Abstraction.
3. Differentiate between a class (the blueprint) and an object (the instance).
4. Design and implement class es with appropriate attributes (data members) and methods (member functions).
5. Apply encapsulation by using access modifiers (like public , private, protected) to control data access.
6. Implement inheritance to create specialized sub-class es from a general super-class , promoting code reuse.
7. Demonstrate polymorphism through method overriding (redefining a method in a subclass) and method overloading (defining multiple methods with the same name but different parameters).
8. Analyze a given problem and design a basic OOP-based solution by identifying potential class es and their relationships.

1.3: OOPS Concepts and Implementation

Object-Oriented Programming (OOP) is a paradigm that has made its way to becoming the most powerful paradigm in the software
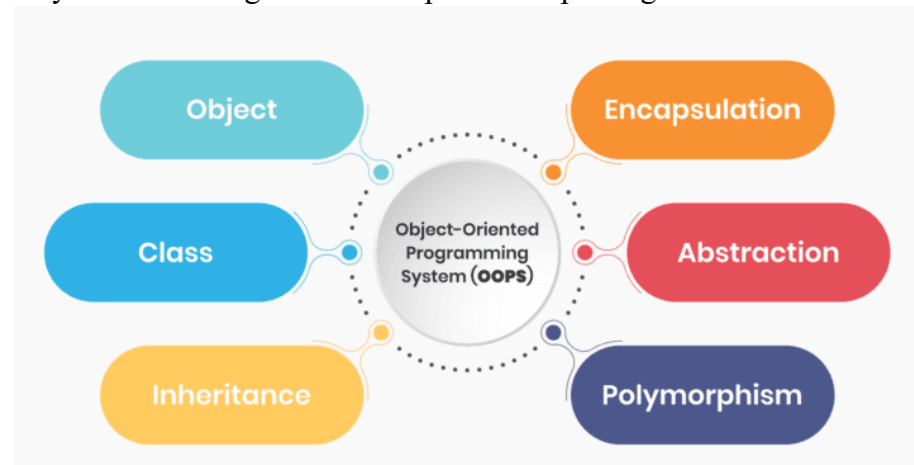
Figure 1.1.: OOPs Pillars

[Source: https://www.colab.research.google.com/

development arena, changing the very notions of how programmers see the world, how they design, and execute the systems. OOP essentially reflects the way in which we view the real world, consisting of a group of individual objects, each with its properties and functions, and among them making meaningful relationships. Java, which first came int o the world in the mid-1990's, has had the reputation as being one of the leading standard-bearers for object-oriented principles, supporting a rich, platform-independent environment which embraces the object-oriented paradigm. While procedural programming is based on functions or the sequence of operations, OOP focuses on objects and methods rather than functions, making them modular. This paradigm transformation cemented OOP as the preeminent approach for designing large, int ricate software solutions across a wide array of sectors, whether enterprise applications, web services, mobile devices, or embedded systems.

The beauty of OOP in Java is that it's organized around six core concepts, namely class es, objects, encapsulation, inheritance, polymorphism and abstraction. However, these principles operate in concert to form a coherent framework that allows developers to model the entities and relations of the real world in their code. Thus, class es are templates that outline both features (attributes) and functionalities (methods) of objects, whilst objects are actual manifestations of these class es, encapsulating their details and bringing them to life in your code. In encapsulation, a protective barrier is set around the object, and the data is restricted from outside access and modification. Inheritance defines the relationships between the base class es and the derived class es. A derived class can also access the members of the base class es. Polymorphism adds a layer of flexibility as an object can use different behavior based on its context, even when derived from common int erfaces. With abstraction, developers can override complexity, working only on properties that are relevant, while hiding behind implementation. These principles form the bedrock on which Java's approach to software development is built, providing developers with a robust arsenal for crafting clean, efficient, and maint ainable code. This in-depth resource goes int o each of the core OOP principles in detail, explaining the theory behind them and how you can apply them in practice in Java. We will explore how these principles manifest in coding practices through step-wise explanation, examples and

applications. When developers understand these principles, they can exploit the full power of Java object orientation and create applications that are not only functional, but also robust, flexible and scalable. This guide is a good fit for you if you are either brand new to programming wanting to get started with object oriented programming using basics of Java or you are a seasoned developer wanting to learn the philosophical point behind the syntax of Java and how Java implements object orientation as a the main paradigm of programming, and what makes it one of the oldest and well built and most used programming languages in the software development industry.

1.4: Class es and Objects

Java is an object-oriented programming language and class es and objects are the building blocks of the object-oriented programming in Java and the framework on which the whole paradigm is based on. Java class : A class in Java is a blueprint or template that defines the properties (attributes) and behaviors (methods) that are common to a particular type of entity It summarizes the core attributes that char acterize what an object is and the actions that specify what an object can do. View a class as an abstract thing—it's the idea of something. For example, a Car class , will have attributes like color, make, model,
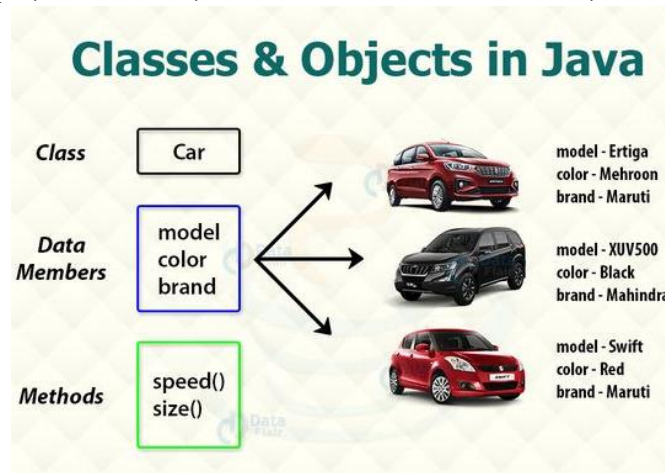
Figure 1.2: Classes and Objects
[Source: https://in.pinterest.com/

and year as well as methods like accelerate(), brake(), and turn(). Note that the class itself does not represent any specific car; it describes the structure that all cars in the program will adhere to. On Java, a class is specified with the Keyword, course, myClass and any code that contains area ideas, constructors, and method definitions. This

organized way of defining a class allows developers to process strong units that truly represent real-world objects.

Class es exist as concepts that define the nature and behavior of objects, while objects are specific occurrences of class es, actual implementations of those ideas.Java implements an object creation concept named instantiation where an object is created with the new keyword followed by calling the constructor method. This action reserves memory space for the object, sets the fields, and returns a reference to the new ly created instance. All fields in the object state independently of all objects in the same class So with one Car Object we might call accelerate () to increase its speed, but another Car Object stays at 0. This allows objects to model separate entities that can collaborate with one another in the program. Class es are the blueprint s for objects; they define the properties and methods that the instantiated objects will have, while objects are the actual entity that is created based on those blueprint s — the things we work with in the program.

This int erplay between class es and objects is what allows Java developers to write modular, organized code that accurately reflects complex systems. By implementing proper OOP principles, programmers are able to group together properties and methods, allowing for the code designed to easily be reused and maint ained. Say for example, in a banking application, the class es could be: Account, Customer, Transaction, Branch. Class es would represent different entities, such as bank accounts, customers, transactions, and branches, and they would define both the properties and behaviors associated with these entities. This allows you to think of the program as a collection of int eracting entities instead of a series of operations, more naturally matching how we approach thinking about systems in the real world. Moreover, the class -object model promotes teamwork across different teams of developers by defining clear boundaries as well as int erfaces between various elements in a system. However, as long as team members follow the contract, they can work on different class es independently, which can provide significant speedups during development of large-scale applications. This clever int erplay of class es and objects grants Java nimbleness and versatility by offering a well-defined framework for creating complex software systems that can grow and evolve over time.

```java
// Example of a class  definition in Java
public      class  Car {
  // Attributes (fields)
  private String   make;
  private String   model;
  private int  year;
  private String   color;
  private double speed;

  // Constructor
  public      Car(String   make, String   model, int  year, String
color) {
    this.make = make;
    this.model = model;
    this.year = year;
    this.color = color;
    this.speed = 0.0;
  }

  // Behaviors (methods)
  public      void    accelerate(double amount) {
    speed += amount;
    System.out.print ln("Car accelerating. Current speed: " + speed +
" mph");
  }

  public      void    brake(double amount) {
    if (speed >= amount) {
      speed -= amount;
    } else {
      speed = 0;
    }
    System.out.print ln("Car braking. Current speed: " + speed + "
mph");
  }

  public      void    turn(String   direction) {
```

```
        System.out.print ln("Car turning " + direction);
}
// Accessor methods (getters)
public      String  getMake() {
    return make;
}

public      String  getModel() {
    return model;
}

public      int  getYear() {
    return year;
}

public      String  getColor() {
    return color;
}

public      double getSpeed() {
    return speed;
}

// Object creation and usage example
public      static void    main(String  [] args) {
    // Creating objects (instances of the Car class )
    Car myCar = new      Car("Toyota", "Camry", 2023, "Red");
    Car friendsCar = new      Car("Honda", "Civic", 2022, "Blue");

    // Using object methods
    System.out.print ln("My car is a " + myCar.getColor() + " " +
            myCar.getYear() + " " + myCar.getMake() +
            " " + myCar.getModel());

    myCar.accelerate(30);
    myCar.turn("right");
    myCar.brake(10);
```

```
        System.out.print ln("Friend's car is a " + friendsCar.getColor() +
" " +
                friendsCar.getYear() + " " + friendsCar.getMake() +
                " " + friendsCar.getModel());


        friendsCar.accelerate(45);
        friendsCar.turn("left");
        friendsCar.brake(15);
    }
}
```

Output:

My car is a Red 2023 Toyota Camry

Car accelerating. Current speed: 30.0 mph

Car turning right

Car braking. Current speed: 20.0 mph

Friend's car is a Blue 2022 Honda Civic

Car accelerating. Current speed: 45.0 mph

Car turning left

Car braking. Current speed: 30.0 mph

Explanation of output:

1. System.out.print ln("My car is a ...") → print s info about myCar.
2. myCar.accelerate(30); → increases speed to 30.0 and print s current speed.
3. myCar.turn("right"); → print s turning right.
4. myCar.brake(10); → reduces speed to 20.0 and print s current speed.
5. System.out.print ln("Friend's car is a ...") → print s info about friendsCar.
6. friendsCar.accelerate(45); → increases speed to 45.0 and print s current speed.
7. friendsCar.turn("left"); → print s turning left.
8. friendsCar.brake(15); → reduces speed to 30.0 and print s current speed.

1.5: Encapsulation

Encapsulation is one of the four core principles of Object-Oriented Programming and signifies the concept of encapsulation where objects hide information and provide controlled access to its int ernal state. In its simplest form, of encapsulation is bundling attributes (data) and the methods that affect those data int o a single entity (class) and restricting access to the int ernal constituents of that entity. This mechanism acts as a wall between the object with a hidden value and code running outside it, the latter running any int erference with an external code trying to meddle with an object's hidden variable. Java encapsulation is
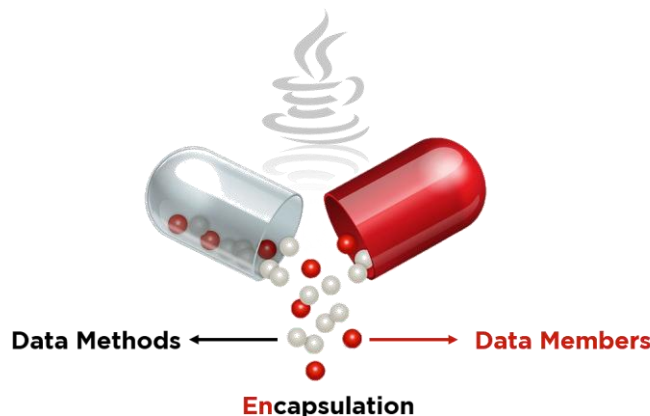
Figure1.2 : Encapsulation
Source: https://www.simplilearn.com/

mainly achieved using access modifiers, which are keywords that determine the visibility or accessibility of a class member (private, protected, and public   ). Private modifier allows code from other class es to access the field only if it is defined in the same class , which makes it an essential tool for encapsulation. This protects the object's char   acters from being accessed by external code directly, and keeps the object's data valid without invalidating its state in its fucking life. Java developers can create more portable and reusable software components through effective encapsulation by separating an object's implementation from its int erface.

Default encapsulation is very simple in Java, for achieving default encapsulation we use encapsulation like if you need we declare class attributes as private and provides public   methods (getters and setters). This strategy has some major benefits in software development. First, it gives the class designer the ability to enforce validation right in the setter methods, making sure that attributes can only be assigned valid values. For example, a setter method for an

employee's salary might check that the new salary value is positive and in a reasonable range before making the change. Second, encapsulation allows int ernal implementation details to change, without having to change any code that uses this class . The public int erface may be the same while changing the int ernal representation of the attribute from some simple primitive type (String , int eger, etc.) to a complex object, thereby allowing keeping the backward compatibility. Third, encapsulation allows for additional logic to be attached to the reading or writing of properties — think of logging a change, notification of observers, or maint aining consistency between related properties. While the contract enforced by this controlled access pattern ensures that systems are more easily predictable and maint ainable over time.

Encapsulation also serves as a guiding principle for software design, ensuring loose coupling and separation of concerns. Encapsulation minimizes int er-component dependencies by hiding implementation details and presenting only necessary int erfaces. This modularity allows different class es to evolve separately so long as they adhere to their contractually specified int erfaces, which also allows for parallel development and incremental modification of large codebases. Encapsulation also enables defensive programming practices by minimizing the exposure of attributes—once they can only be changed through cleanly defined methods, the places where bugs might creep in are limited and hence can be easily located. Encapsulation also implements the principle of least privilege in software design, which ensures information is accessible only on a need-to-know basis. Limiting access rights reduces the risk of security vulnerabilities and side effects in complicated systems. By providing such a wide variety of advantages, encapsulation become a core principle of Java programming, empowering developers to build software that is not merely functional, but also secure, maint ainable, and adaptable to changing needs.

```
// Example of encapsulation in Java
public      class  BankAccount {
    // Private attributes - hidden from outside access
    private String   accountNumber;
    private String   accountHolderName;
```

```java
    private double balance;
    private String   accountType;
    private boolean isActive;

    // Constructor
    public        BankAccount(String   accountNumber, String
accountHolderName, double initialDeposit, String   accountType) {
        this.accountNumber = accountNumber;
        this.accountHolderName = accountHolderName;
        this.balance = initialDeposit;
        this.accountType = accountType;
        this.isActive = true;
    }

    // Getter methods - controlled access to private attributes
    public        String   getAccountNumber() {
        // Return masked account number for security
        return "XXXX-XXXX-" + accountNumber.subString
(accountNumber.length() - 4);
    }

    public        String   getAccountHolderName() {
        return accountHolderName;
    }

    public        double getBalance() {
        return balance;
    }

    public        String   getAccountType() {
        return accountType;
    }

    public        boolean isActive() {
        return isActive;
    }

    // Setter methods - controlled modification with validation
```

```java
public void setAccountHolderName(String
accountHolderName) {
    if (accountHolderName != null &&
!accountHolderName.trim().isEmpty()) {
        this.accountHolderName = accountHolderName;
    } else {
        throw new     IllegalArgumentException("Account holder
name cannot be empty");
    }
}


// No setter for account number - it should not be changed after
creation

public void setAccountType(String accountType) {
    if (accountType != null && (accountType.equals("Checking") ||
                accountType.equals("Savings") ||
                accountType.equals("Investment"))) {
        this.accountType = accountType;
    } else {
        throw new     IllegalArgumentException("Invalid account
type. Must be Checking, Savings, or Investment");
    }
}

public void setActive(boolean isActive) {
    this.isActive = isActive;
}

// Business methods that modify the private attributes in a
controlled way
public void deposit(double amount) {
    if (!isActive) {
        throw new     IllegalStateException("Cannot deposit to
inactive account");
    }

    if (amount <= 0) {
```

```
        throw new      IllegalArgumentException("Deposit amount
must be positive");
    }

    balance += amount;
    System.out.print ln("Deposited: $" + amount + ". New
balance: $" + balance);
  }

  public     void    withdraw(double amount) {
    if (!isActive) {
      throw new      IllegalStateException("Cannot withdraw from
inactive account");
    }

    if (amount <= 0) {
      throw new      IllegalArgumentException("Withdrawal
amount must be positive");
    }

    if (amount > balance) {
      throw new      IllegalStateException("Insufficient funds");
    }

    balance -= amount;
    System.out.print ln("Withdrawn: $" + amount + ". New
balance: $" + balance);
  }

  // Example usage
  public     static void    main(String  [] args) {
    BankAccount account = new      BankAccount("1234567890",
"John Doe", 1000.0, "Checking");

    // Access attributes through getters
    System.out.print ln("Account: " +
account.getAccountNumber());
```

```
        System.out.print ln("Holder: " +
account.getAccountHolderName());
        System.out.print ln("Balance: $" + account.getBalance());
        System.out.print ln("Type: " + account.getAccountType());

        // Modify attributes through setters and business methods
        account.setAccountHolderName("John A. Doe");
        account.deposit(500);
        account.withdraw(200);

        // This would throw an exception:
        // account.balance = -1000; // Compilation error: balance is
private

        // Using methods with validation
        try {
            account.withdraw(2000); // Will throw exception for
insufficient funds
        } catch (IllegalStateException e) {
            System.out.print ln("Error: " + e.getMessage());
        }
    }
}
```

If you compile and run your **BankAccount** program, you will see:

Output:
Account: XXXX-XXXX-7890
Holder: John Doe
Balance: $1000.0
Type: Checking
Deposited: $500.0. New   balance: $1500.0
Withdrawn: $200.0. New   balance: $1300.0
Error: Insufficient funds

Explanation:
  1. **First four print ln statements:**

- o getAccountNumber() masks all but the last 4 digits ⇒ XXXX-XXXX-7890
- o getAccountHolderName() ⇒ John Doe
- o getBalance() ⇒ 1000.0
- o getAccountType() ⇒ Checking

2. **account.setAccountHolderName("John A. Doe");**

✔ Changes the name int ernally but does not print anything.

3. **account.deposit(500);**

✔ Adds $500 ⇒ balance becomes $1500 ⇒ print s:

Deposited: $500.0. New   balance: $1500.0

4. **account.withdraw(200);**

✔ Subtracts $200 ⇒ balance becomes $1300 ⇒ print s:

Withdrawn: $200.0. New   balance: $1300.0

5. **account.withdraw(2000); inside try block**

 Triggers Insufficient funds exception ⇒ caught by catch block ⇒ print s:

Error: Insufficient funds

Note:

Directly modifying balance from outside would cause a **compilation error** because balance is private.

You can only int eract through the provided methods—this is **encapsulation** in action.

1.6: Inheritance
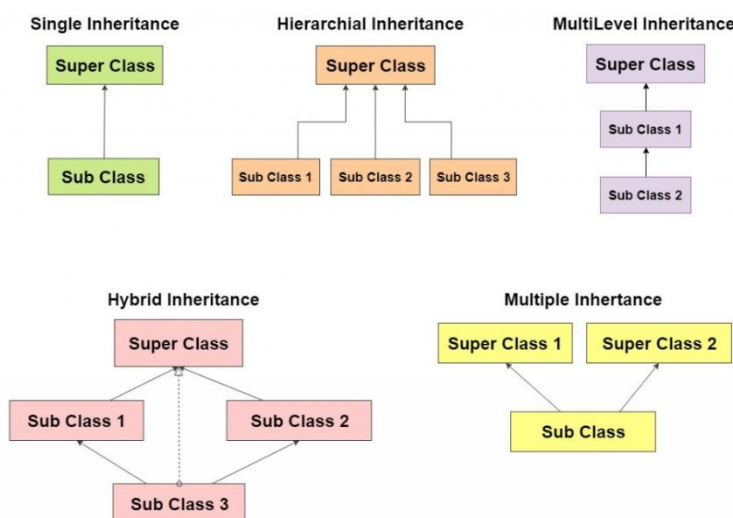
Figure1.3: Types of Inheritance

One of the fundamental building blocks of object-oriented programming in Java is inheritance, which allows developers create a relationship between class es that follows the same "is-a" relationship found in real-world taxonomies. This is a potent process, enabling developers to create a new   class (the subclass or derived class) that extends an existing class (the superclass or base class) to inherit its char   acteristics, and functionality, while providing new   or altered functionality where required. The extends keyword in Java is used to implement inheritance, forming a parent-child relationship between class es, where the child class   automatically inherits all the visible members (fields and methods) from its parent class . This relationship defines inheritance of common attributes and behaviors, which, due to dynamic polymorphism, can be defined only once, in a parent class , and reused in multiple child class es. An example of a Vehicle class   could provide common attributes such as speed, color, and weight, along with methods to start, stop, and rate fuel consumption. Language class es like Car, Motorcycle, and Truck can inherit these common properties, but they can also int roduce their own specific properties, like the number of doors for the car or the capacity of a truck. This simple hierarchy eliminates code duplication, but also creates a natural organization that follows the conceptual relationships between different types of entities.

The inheritance model of Java has some unique features that helps the developers design their class hierarchies. Because Java only supports single inheritance for class es — that is, it only allows a class to extend one superclass — this helps avoid   the complexities and ambiguities associated with multiple inheritance. Java, for instance, compensates for this limitation with int erfaces, permitting a class to form a contract with multiple int erfaces, effectively creating a sort of multiple inheritance of behavior. Second, the super keyword in Java refers to the superclass , allowing subclass es to access inherited methods and call superclass constructors. This ensures that the inherited fields are initialized appropriately, and it also enables subclass es to build upon and broaden the behaviors specified in their parent class . 3. Java's model of inheritance provides the concept of method overriding, where a subclass implements a specialized version of a method defined in its superclass . The purpose of the @Override annotation is to inform the

compiler that the annotated method is being overridden from its superclass , allowing it to check whether the method signature matches an inherited method and providing an error in case of method overloading. You are vertical after October twenty twenty-three.

Inheritance in Java, however, is a philosophical approach to the design of programs that centers on the concepts of generalization and specialization, beyond the technical side. With inheritance, developers can formulate abstract base class es that encapsulate the core properties of a concept, and then derive specialized subclass es that adapt and add to this concept for common scenarios. This maps very nicely to how we humans experience and group our knowledge, making for more sensible and natural object-oriented designs. There are things like polymorphic behavior, where a collection of objects of various subclass types can be handled uniformly via their common superclass type, enabling greater flexibility and extensibility in software systems. The point is that if you have a class with a method that takes a Shape, you can call that method with any Circle, Rectangle, or Triangle subclass ed object, since they all inherit—from some other class (directly or indirectly) from Shape. This polymorphic feature allows programmers to write code that works with existing types and future derivatives without needing to change the code, providing an ideal case of the open-closed principle in software design. Moreover, inheritance enables incremental development and testing, as base class es can be implemented and validated prior to the addition of derived class es. Inheritance continues to be a vital concept within the Java programming language, allowing developers to create software architectures that are both structurally sound and responsive to changing needs by providing its various advantages.

```java
// Example of inheritance in Java
// Base class  (superclass )
public      class Vehicle {
    // Common attributes for all vehicles
    protected String   brand;
    protected String   model;
    protected int  year;
    protected double speed;
    protected double fuelCapacity;
```

```java
    protected double fuelLevel;

// Constructor
public    Vehicle(String  brand, String  model, int  year, double
fuelCapacity) {
    this.brand = brand;
    this.model = model;
    this.year = year;
    this.speed = 0;
    this.fuelCapacity = fuelCapacity;
    this.fuelLevel = fuelCapacity / 2; // Start with half tank
}

// Common behaviors for all vehicles
public    void    start() {
    System.out.print ln("Vehicle starting...");
}

public    void    stop() {
    speed = 0;
    System.out.print ln("Vehicle stopped.");
}

public    void    accelerate(double amount) {
    if (fuelLevel > 0) {
        speed += amount;
        consumeFuel(amount * 0.1); // Simple fuel consumption
model
        System.out.print ln("Vehicle accelerating. Current speed: " +
speed + " mph");
    } else {
        System.out.print ln("Cannot accelerate. Out of fuel.");
    }
}

public    void    refuel(double amount) {
    if (fuelLevel + amount <= fuelCapacity) {
        fuelLevel += amount;
```

```java
        } else {
            fuelLevel = fuelCapacity;
        }
        System.out.print ln("Refueled. Current fuel level: " + fuelLevel +
" gallons");
    }

    protected void    consumeFuel(double amount) {
        fuelLevel = Math.max(0, fuelLevel - amount);
        if (fuelLevel == 0) {
            System.out.print ln("Warning: Vehicle out of fuel!");
        }
    }

    // Getters
    public      String   getBrand() { return brand; }
    public      String   getModel() { return model; }
    public      int  getYear() { return year; }
    public      double getSpeed() { return speed; }
    public      double getFuelLevel() { return fuelLevel; }

    @Override
    public      String   toString () {
        return year + " " + brand + " " + model;
    }
}

// Derived class  (subclass )
public      class  Car extends Vehicle {
    // Additional attributes specific to cars
    private int  numberOfDoors;
    private boolean hasConvertibleTop;
    private boolean isTrunkOpen;

    // Constructor that calls the superclass  constructor
    public      Car(String  brand, String  model, int  year, double
fuelCapacity, int  numberOfDoors, boolean hasConvertibleTop) {
```

```java
        super(brand, model, year, fuelCapacity); // Call to superclass
constructor
        this.numberOfDoors = numberOfDoors;
        this.hasConvertibleTop = hasConvertibleTop;
        this.isTrunkOpen = false;
    }

    // Override the start method from Vehicle
    @Override
    public      void     start() {
        System.out.print ln("Car engine starting... Vroom!");
        super.start(); // Call the superclass  version of the method
    }

    // Additional behaviors specific to cars
    public      void     openTrunk() {
        isTrunkOpen = true;
        System.out.print ln("Car trunk opened.");
    }

    public      void     closeTrunk() {
        isTrunkOpen = false;
        System.out.print ln("Car trunk closed.");
    }

    public      void     toggleConvertibleTop() {
        if (hasConvertibleTop) {
            System.out.print ln(hasConvertibleTop ? "Convertible top
opened." : "Convertible top closed.");
        } else {
            System.out.print ln("This car doesn't have a convertible top.");
        }
    }

    // Override the toString   method from Vehicle
    @Override
    public      String  toString () {
        return super.toString () + " (Car, " + numberOfDoors + "-door" +
```

```
                (hasConvertibleTop ? ", Convertible" : "") + ")";
    }


    // Getters for car-specific attributes
    public      int  getNumberOfDoors() { return numberOfDoors; }
    public      boolean hasConvertibleTop() { return
hasConvertibleTop; }
    public      boolean isTrunkOpen() { return isTrunkOpen; }
}


// Another derived class  showing inheritance
public      class  Motorcycle extends Vehicle {
    // Additional attributes specific to motorcycles
    private boolean hasSideCar;
    private String   engineType;


    // Constructor
    public      Motorcycle(String   brand, String   model, int  year,
double fuelCapacity, boolean hasSideCar, String   engineType) {
        super(brand, model, year, fuelCapacity);
        this.hasSideCar = hasSideCar;
        this.engineType = engineType;
    }


    // Override the start method
    @Override
    public      void     start() {
        System.out.print ln("Motorcycle engine starting... Rumble!");
        super.start();
    }


    // Override the accelerate method for different fuel consumption
    @Override
    public      void     accelerate(double amount) {
        if (fuelLevel > 0) {
            speed += amount * 1.5; // Motorcycles accelerate faster
            consumeFuel(amount * 0.05); // Motorcycles use less fuel
```

```java
            System.out.print ln("Motorcycle accelerating. Current speed: "
+ speed + " mph");
        } else {
            System.out.print ln("Cannot accelerate. Out of fuel.");
        }
    }

    // Additional methods specific to motorcycles
    public    void    performWheelie() {
        if (speed > 15) {
            System.out.print ln("Performing a wheelie! Be careful!");
        } else {
            System.out.print ln("Speed too low for a wheelie.");
        }
    }

    // Override toString
    @Override
    public    String   toString () {
        return super.toString () + " (Motorcycle, " + engineType + "
engine" +
            (hasSideCar ? " with sidecar" : "") + ")";
    }

    // Getters
    public    boolean hasSideCar() { return hasSideCar; }
    public    String   getEngineType() { return engineType; }
}

// Example usage
public    class  InheritanceDemo {
    public    static void    main(String  [] args) {
        // Create objects of different vehicle types
        Vehicle genericVehicle = new      Vehicle("Generic",
"Transporter", 2023, 15.0);
        Car sedan = new      Car("Toyota", "Camry", 2023, 14.5, 4,
false);
```

```
        Car convertible = new      Car("Mazda", "MX-5", 2023, 11.9, 2,
true);
        Motorcycle sportBike = new      Motorcycle("Honda",
"CBR600RR", 2023, 4.5, false, "4-cylinder");

        // Demonstrate inheritance by using common methods
        System.out.print ln("\n--- Generic Vehicle ---");
        System.out.print ln(genericVehicle);
        genericVehicle.start();
        genericVehicle.accelerate(30);
        genericVehicle.stop();

        System.out.print ln("\n--- Sedan ---");
        System.out.print ln(sedan);
        sedan.start();
        sedan.accelerate(35);
        sedan.openTrunk();
        sedan.closeTrunk();
        sedan.stop();

        System.out.print ln("\n--- Convertible ---");
        System.out.print ln(convertible);
        convertible.start();
        convertible.accelerate(40);
        convertible.toggleConvertibleTop();
        convertible.stop();

        System.out.print ln("\n--- Sport Bike ---");
        System.out.print ln(sportBike);
        sportBike.start();
        sportBike.accelerate(50);
        sportBike.performWheelie();
        sportBike.stop();

        // Demonstrate polymorphism (will be covered in more detail in
the polymorphism section)
        System.out.print ln("\n--- Polymorphic Behavior ---");
```

```
        Vehicle[] vehicles = {genericVehicle, sedan, convertible,
sportBike};

        for (Vehicle v : vehicles) {
            System.out.print ln("Processing: " + v);
            v.start();
            v.accelerate(25);
            v.stop();
            System.out.print ln();
        }
    }
}
```

Output:

--- Generic Vehicle ---

2023 Generic Transporter

Vehicle starting...

Vehicle accelerating. Current speed: 30.0 mph

Vehicle stopped.

--- Sedan ---

2023 Toyota Camry (Car, 4-door)

Car engine starting... Vroom!

Vehicle starting...

Vehicle accelerating. Current speed: 35.0 mph

Car trunk opened.

Car trunk closed.

Vehicle stopped.

--- Convertible ---

2023 Mazda MX-5 (Car, 2-door, Convertible)

Car engine starting... Vroom!

Vehicle starting...

Vehicle accelerating. Current speed: 40.0 mph

Convertible top opened.

Vehicle stopped.

--- Sport Bike ---

2023 Honda CBR600RR (Motorcycle, 4-cylinder engine)

Motorcycle engine starting... Rumble!

Vehicle starting...

Motorcycle accelerating. Current speed: 75.0 mph

Performing a wheelie! Be careful!

Vehicle stopped.

--- Polymorphic Behavior ---

Processing: 2023 Generic Transporter

Vehicle starting...

Vehicle accelerating. Current speed: 25.0 mph

Vehicle stopped.

Processing: 2023 Toyota Camry (Car, 4-door)

Car engine starting... Vroom!

Vehicle starting...

Vehicle accelerating. Current speed: 60.0 mph

Vehicle stopped.

Processing: 2023 Mazda MX-5 (Car, 2-door, Convertible)

Car engine starting... Vroom!

Vehicle starting...

Vehicle accelerating. Current speed: 65.0 mph

Vehicle stopped.

Processing: 2023 Honda CBR600RR (Motorcycle, 4-cylinder engine)

Motorcycle engine starting... Rumble!

Vehicle starting...

Motorcycle accelerating. Current speed: 112.5 mph

Performing a wheelie! Be careful!

Vehicle stopped.

Explanation:
- **toString () overrides** provide readable descriptions (year brand model …).
- Each start() method print s a message, with subclass es adding their own output before calling super.start().
- accelerate() print s updated speeds and consumes fuel:

- o Vehicle accelerates normally.
- o Motorcycle accelerates 1.5× faster.
- Car-specific methods (openTrunk, toggleConvertibleTop) print their actions.
- performWheelie() in Motorcycle checks speed before print ing.
- The polymorphism loop (Vehicle[]) calls overridden methods dynamically for each subclass .

The polymorphism loop (Vehicle[]) calls overridden methods dynamically for each subclass .

1.7: Polymorphism

When something may illustrate the measure of one thing, polymorphism, from the Greek words significance "many forms," is viewed as a standout amongst the most influential concepts in object-situated programming you can have diverse items at different circumstances to a similar int erface in different ways.
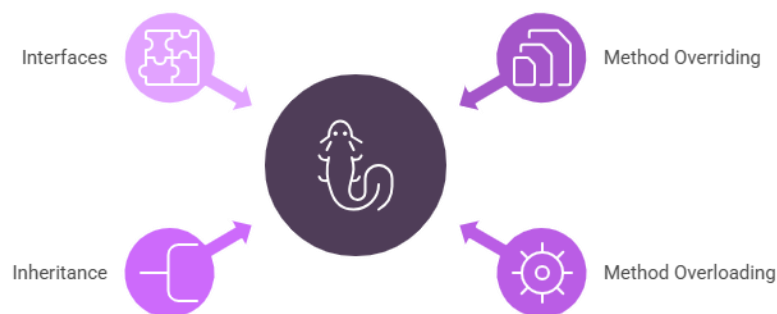
Figure1.5: Elements of Polymorphism

Polymorphism in Java is mainly achieved through method overriding and method overloading, providing a flexibility towards writing a more elegant and extensible code flow. In simpler terms, when the subclass has the same method as its super class , we call this method as method overriding and thus subclass method will be called while invoking the method on a class object. This dynamic method dispatch, also referred to as runtime polymorphism, is based on the actual type of the object rather than the reference type. Example: If we have a superclass reference point ing to a subclass object and call a method we would expect from the superclass to be called Java would automatically invoke the one overridden from the subclass . Method overloading,

however, is an example of compile-time polymorphism, because it defines multiple methods with the same name but different argument lists to exist in the same class . The Java compiler decides which version of the method should be execute based on number of arguments, types of arguments and order of arguments passed. In combination, but with the aid of such mechanisms, Java developers can implement code that operates on objects at increasing levels of abstraction whereby they are manipulated through common int erfaces while their specific implementations can still vary, which provides the user reusability of code blocks and simplifies the evolution of the system.

Polymorphism in Java, which would be the basis of this article, in practical terms, is only deriving from the int erplay of the concepts of inheritance and int erfaces. Case in point , through inheritance, subclass es can override any methods declared in their superclass es, allowing you to provide specialized behavior while preserving the method signature. It allows code in the client to communicate with objects using superclass reference variables, treating heterogeneous cases of object types uniformly, according to common inheritance. An example would be a drawing application that creates a Shape superclass with Circle, Rectangle and Triangle subclass es. Client code on a Shape reference doesn't need to know its subtype, it can just call draw(), and each subclass knows its rendering logic, override draw(). So int erfaces take this polymorphic capability to the next level by defining contracts that different class es must implement. A class can implement many int erfaces, where methods of the int erface describe different aspects of its behavior, enabling objects to be treated polymorphically based on their abilities rather than their inheritance lineage. For example, unrelated class es such as ElectricCar, SolarPanel, and Smartphone might all implement a common Rechar geable int erface which would allow them to be processed in a consistent way by healing systems. This late binding that is made possible by showing this int erface-based polymorphic behaviour enables us to develop systems with high degree of flexibility since new types can just be added without the need of making any changes to existing code.

This goes beyond its technical application: polymorphism is a way of thinking about software, a philosophy of design that's focused around

abstraction and behavior-oriented design. Polymorphism, by emphasizing that it is what objects do and not what objects are that matters, encourages developers to design systems around behavior and capabilities, leading to more flexible, loosely coupled architectures. This also enables the open-closed principle, which states that software entities should be open to extension but closed to modification, also allowing systems to evolve in a way that new implementations can be registered, rather than modifying existing code. When implemented correctly, polymorphism also supports the strategy pattern and other behavioral design patterns where an algorithm is chosen based on context at runtime. As an example, a navigation system could implement a different pathfinder algorithm (all implementing the same RouteStrategy int erface) depending on whether the user prefers the fastest, the most scenic, or the most fuel-efficient route. Such a dynamic behavior makes the applications more responsive and aware of the context. Moreover, polymorphism leads to more int entional code as methods can retain the same name in different implementations, aligning themselves with the conceptual idea rather than the implementation.

```java
// Example of polymorphism in Java
// Base int erface defining a common behavior
public      int erface Shape {
   double calculateArea();
   double calculatePerimeter();
   void    draw();
   String  getType();
}

// Concrete implementation of Shape: Circle
public      class  Circle implements Shape {
   private double radius;

   public      Circle(double radius) {
      this.radius = radius;
   }

   @Override
```

```java
    public        double calculateArea() {
        return Math.PI * radius * radius;
    }


    @Override
    public        double calculatePerimeter() {
        return 2 * Math.PI * radius;
    }
    @Override
    public        void    draw() {
        System.out.print ln("Drawing a circle with radius " + radius);
        // Imagine more complex drawing logic here
    }


    @Override
    public        String  getType() {
        return "Circle";
    }


    // Circle-specific method
    public        double getDiameter() {
        return 2 * radius;
    }
}

// Concrete implementation of Shape: Rectangle
public        class  Rectangle implements Shape {
    private double length;
    private double width;

    public        Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    @Override
    public        double calculateArea() {
        return length * width;
```

```
    }

    @Override
    public      double calculatePerimeter() {
        return 2 * (length + width);
    }

    @Override
    public      void    draw() {
        System.out.print ln("Drawing a rectangle with length " + length
+ " and width " + width);
        // Imagine more complex drawing logic here
    }
    @Override
    public      String   getType() {
        return "Rectangle";
    }

    // Rectangle-specific method
    public      boolean isSquare() {
        return length == width;
    }
}

// Concrete implementation of Shape: Triangle
public      class  Triangle implements Shape {
    private double sideA;
    private double sideB;
    private double sideC;

    public      Triangle(double sideA, double sideB, double sideC) {
        // Validate that the sides can form a triangle
        if (sideA + sideB <= sideC || sideA + sideC <= sideB || sideB +
sideC <= sideA) {
            throw new      IllegalArgumentException("The sides do not
form a valid triangle");
        }
```

```java
      this.sideA = sideA;
      this.sideB = sideB;
      this.sideC = sideC;
   }

   @Override
   public    double calculateArea() {
      // Heron's formula
      double s = (sideA + sideB + sideC) / 2;
      return Math.sqrt(s * (s - sideA) * (s - sideB) * (s - sideC));
   }

   @Override
   public    double calculatePerimeter() {
      return sideA + sideB + sideC;
   }

   @Override
   public    void    draw() {
      System.out.print ln("Drawing a triangle with sides " + sideA + ",
" + sideB + ", and " + sideC);
      // Imagine more complex drawing logic here
   }

   @Override
   public    String  getType() {
      return "Triangle";
   }

   // Triangle-specific method
   public    boolean isEquilateral() {
      return sideA == sideB && sideB == sideC;
   }
}

// Demo class  to show polymorphism
public     class  PolymorphismDemo {
   public     static void    main(String  [] args) {
```

```
    // Polymorphic collection
    Shape[] shapes = {
        new     Circle(5),
        new     Rectangle(4, 6),
        new     Triangle(3, 4, 5),
        new     Rectangle(5, 5),
        new     Circle(2.5)
    };

    // Process shapes in a polymorphic way
    for (Shape shape : shapes) {
        System.out.print ln("\nShape Type: " + shape.getType());
        shape.draw();
        System.out.print ln("Area: " + shape.calculateArea());
        System.out.print ln("Perimeter: " +
shape.calculatePerimeter());
    }
  }
}
```

Code demonstration:

- Shape int erface defines a common contract (calculateArea, calculatePerimeter, draw, getType).
- Circle, Rectangle, Triangle each implement Shape and override methods with their own logic.
- A polymorphic array of Shape allows you to call the same methods on different shapes without knowing their concrete types at compile time.
- Each shape responds with its own implementation (runtime polymorphism).

Output

Shape Type: Circle

Drawing a circle with radius 5.0

Area: 78.53981633974483

Perimeter: 31.41592653589793

Shape Type: Rectangle

Drawing a rectangle with length 4.0 and width 6.0

Area: 24.0

Perimeter: 20.0

Shape Type: Triangle

Drawing a triangle with sides 3.0, 4.0, and 5.0

Area: 6.0

Perimeter: 12.0

Shape Type: Rectangle

Drawing a rectangle with length 5.0 and width 5.0

Area: 25.0

Perimeter: 20.0

Shape Type: Circle

Drawing a circle with radius 2.5

Area: 19.634954084936208

Perimeter: 15.707963267948966

Check Your Progress

1. **What is the difference between a class and an object?.**

   ………………………………………………………………
   ………………………………………………………………
   ………………………………………………………………
   ………………………………………………………………

2. **List the four main principles of Object-Oriented Programming.**

   ………………………………………………………………
   ………………………………………………………………
   ………………………………………………………………
   ………………………………………………………………

1.8: Summary

This unit provided a essential foundation in Object-Oriented Programming (OOP) as it applies to Java. We learned that OOP is a model based on objects, which are runtime instances of classes. A class acts as a blueprint, defining attributes (data) and methods (behavior). The paradigm's strength comes from its four pillars: Encapsulation,

which bundles data and methods securely; Abstraction, which simplifies complex systems by hiding unnecessary details; Inheritance, which promotes code reuse by allowing classes to inherit properties from others; and Polymorphism, which provides flexibility by allowing actions to be processed differently based on the object. These concepts work together to help developers create complex applications that are easier to scale, debug, and maintain.

1.9: Exercises

Multiple Choice Questions:

1. Which of the following best defines Object-Oriented Programming?
   a) A method of using functions to process data sequentially
   b) A paradigm where data and methods are encapsulated within objects
   c) A style focusing only on procedural logic
   d) A technique for file management in systems
   Answer: b) A paradigm where data and methods are encapsulated within objects

2. Which of the following is NOT a feature of OOP?
   a) Inheritance
   b) Encapsulation
   c) Polymorphism
   d) Compilation
   Answer: d) Compilation

3. The process of hiding internal details of an object is known as:
   a) Polymorphism
   b) Encapsulation
   c) Inheritance
   d) Abstraction
   Answer: b) Encapsulation

4. In Java, polymorphism allows:
   a) The creation of multiple unrelated class es
   b) Methods to perform differently based on the calling object
   c) Variables to store multiple types of data
   d) Objects to contain only data and no methods
   Answer: b) Methods to perform differently based on the calling object

5. Abstraction in OOP helps to:

a) Reveal implementation details

b) Hide complex details while showing essential features

c) Reduce runtime errors

d) Increase code length

Answer: b) Hide complex details while showing essential features

Descriptive Questions:

1. Explain the key features of Object-Oriented Programming with examples.
2. Describe how inheritance and polymorphism support code reusability.
3. Compare OOP with procedural programming.
4. Discuss real-world applications where OOP principles are effectively implemented.

1.10: References and Suggested Readings

- Balagurusamy, E. (2020). *Programming with Java*. McGraw Hill.

- Schildt, H. (2018). *Java: The complete reference* (10th ed.). McGraw Hill Education.

- Horstmann, C. S. (2021). *Core Java, Volume I: Fundamentals* (12th ed.). Pearson.

Block 1: Object-Oriented Programming Concepts and Implementations

Unit 2: Package Concepts and Implementation

Structure

2.1: Introduction

As Java applications grow in size, a simple list of class files becomes unmanageable. Packages are Java's primary mechanism for organizing code into logical groups. Think of them as "folders" or namespaces for your classes. A package groups related classes and interfaces, helping to prevent naming conflicts. For example, you might place all your database-related classes in a com.myapp.database package and all your user interface classes in a com.myapp.ui package. This unit covers how to create, name, and use packages, including the important import

keyword, which allows you to access classes from other packages, and the concept of access control, which determines which classes can see and use each other.

2.2: Learning Outcomes

After completing this unit, you will be able to:

1. Explain the purpose and benefits of using packages in Java.

2. Declare a package in a Java source file using the package keyword.

3. Import classes from other packages using the import statement.

4. Understand the directory structure that Java uses to mirror the package hierarchy.

5. Describe the role of access modifiers (public, protected, default, private) in relation to package visibility.

2.3: Package Concepts and Implementation

The Java language has a great mechanism the packages for systematic programming. In Java, a package is a namespace that organizes a set of related class es, int erfaces, and sub-packages. The package concept

Figure: 2.1 Key Concept of Packages

is the one of the fundamental concepts of Java that helps rich developers to organize their largescale applications by grouping related components.

Packages have several roles in the Java ecosystem: They help prevent naming conflicts and control access to class es and their members, as well as allow for grouping of related code int o logical units. As an object-oriented language, Java uses packages to allow one to group class es in a logical manner and promotes modular programming approaches that help with code maint ainability, reuse, and scalability — all of which are essential in the development of enterprise-grade applications. The core API itself is organized int o packages, and the Java platform itself is based on packages (e.g., java. lang, java. util, and java. io — each containing class es for specific functionality. I recommend this tutorial for those int erested in learning how to use packages when developing software in Java. As applications scale in size and complexity this organization only becomes more important. Moreover, packages form the basis for the access control mechanism in Java's security model. This Unit delves int o the theoretical constructs of Java packages, analyzes the implementation aspects, and offers practical advice for utilizing packages efficiently in Java development. We will take a closer look at the syntax to define packages, access permissions for package members, compilation and execution of the code organized in packages, and good practices to design packages. Therefore, by applying concepts of Java packages and using them, a developer can have a more maint ainable, secured, and professional structure of code so that with the increasing complexity of the application, one can scale accordingly.

2.4: The Purpose of Packages

Java follow on packages which address certain issues faced in software development. Allowing packages, one of the main advantages to packages are that they provide a method for grouping sets of class es and int erfaces together, which apply some structure to the code that is hierarchical and reflects the logical structure of an application. Plus, this organizational aspect becomes more and more invaluable as projects increase in scope and complexity. If we did not have packages, all class es would be in one single namespace — which makes working in a codebase really difficult, as well as increasing the chances of naming conflicts. For an enterprise application with hundreds or thousands of class es, organizing them int o logical packages makes the codebase navigable and comprehendible. Namespace management

is another key purpose of packages. Packages: Java uses packages to create a unique namespace for each Java class name to avoid name collisions. Double UsageDevelopers working in parallel on different components of an application can both int roduce a class called Configuration. If these are in different packages (e.g. com. company. ui. Configuration and com. company. database. As their fully qualified names differ, they can coexist without conflict (e.g., if you have a class called App\Configuration, then you can have a class called Some\Other\Configuration). They also allow access control via Java's access modifiers. The default access level in Java, also known as "package-private", limits access to class es and members of the same package. This allows developers to encapsulate implementation details, only exposing what is necessary to the outside world while keeping int ernal workings. This encapsulation is a core tenet of object-oriented design that packaging helps enforce. Furthermore, the packages facilitate Java application deployment and distribution via JAR (Java Archive) files. A JAR file can have several packages and the packages' structure is retained in the jar archive. That greatly simplifies the sharing of Java libraries and applications as standalone packages. So packages are also a part of java security model. The Loosely Couple Package: The Java security manager enforces security policies based on package boundaries, determining what class es from different packages are permitted to perform which operations. Packages recast this level of organization to give more than single applications. Now the conventional reversed domain names (like com. company. (for each organization) and (for each project) enables global unique naming of packages from different organizations and different projects, which eases importing any code from any place.

2.5: Historical Context and Evolution

In this article, we will cover the core concepts of Java packages and their evolution — from static imports to the new modular system that was int roduced with Java 9. When Sun Microsystems shipped Java tools in 1995, packages were already one of the language's built-in features, illustrating the language designers' understanding that structuring code would be key to building scalable applications. To begin with, a simple notion of packages was used for basic namespace management and access control. Before Java 2, the standard library was

smaller, limited to the key package's java. lang, java. util, and java. io. The Java standard library went through some dramatic expansion as Java matured up through versions 1.1, 1.2 (Java 2) and beyond, and packages quickly became critical for organizing the increasingly broad API. Enterprise Applications and the java: namespace with the advent of the Java 2 Platform, Enterprise Edition (J2EE, later Jakarta EE), the associative relationship between packages and namespaces became even stronger. The complexity of Java applications increased drastically around this time, as J2EE applications could now be large and could require multiple teams to work on different components. Packages gave you the necessary structure to manage such complexity. Java packages took a big leap forward with the addition of the Java Package Manager (JPM) and subsequently in Java 9 with the addition Project Jigsaw (Java Module System). This had ameliorated shortcoming of the original package system in areas of dependency management and at a higher level of encapsulation than packages. While packages offered namespace management and rudimentary access control, they did not support declarative dependencies or robust encapsulation boundaries, a capability int roduced in Java 9. In particular, while Java 9 modules build on the existing notion of packages to enable explicit declaration of dependencies and better encapsulation with in a module, they also build on the idea of a module being a discrete unit with its own metadata that control its usage patterns. However, packages are still essential to the organization of Java code. The module system is an addition rather than a replacement to packages, as modules contain packages which in turn contain class es. By learning about the history of packages, developers can better appreciate their place in the history of Java's evolution and their best practices for their use in writing modern Java applications. For all this evolution, the essential syntax and semantics of packages have remained curiously stable throughout this series, meaning code written for early versions of the language will be compatible with contemporary Java environments. Backward compatibility: As new features and capabilities have been added to the language and platform; backward compatibility has been ensured as a faithful promise of Java's design philosophy.

2.6: Package Declaration and Naming Conventions

The syntax used for declaring packages in Java is simple, as it sets up the namespace for the class es and int erfaces that the package contains. These class es and int erfaces in a Java source file belong to the specified package and must always be the first non-comment statement in the Java source file. Essentially, usage of a package declaration would look something like this: package packageName; here packageName should follow the naming conventions of Java. If a class does have an explicit package declaration, it belongs to the default package, an unnamed package that has neither of the organizational and access control benefits that named packages provide. For instance, birth a class to belong to a package called "utilities", the first line of the source file may read as follows: package utilities; The package name generally used in Java is hierarchical in nature and compulsory providing some forms of global uniqueness within the code area as it is related to the organization structure. Names for packages are based on reversed domain names, followed by disambiguating identifiers that further constrain the scope. For example, giving a utility class by a company with the domain "example. A package which contains a class from a com. example. utilities; It can be extended, in the logical level, to represent project name, modules and specific functionality: package com. example. projectname. module. feature The package declaration establishes a direct correspondence between the package name and the directory structure that the Java source files are organized. For the package declarationpackage com. example. the corresponding source file has to be placed in the directory needs to match package structure: /com/example/utilities/ It is worth noting that package names must also match the actual directory structure of the source files, and this is enforced at the Java compiler level and is critical to having meaningful package semantics.

2.7: Package Naming Conventions and Standards

The Java™ naming conventions for packages have grown from the needs of development of large software. The reverse domain name

convention is the most widely followed convention that prefix package names.
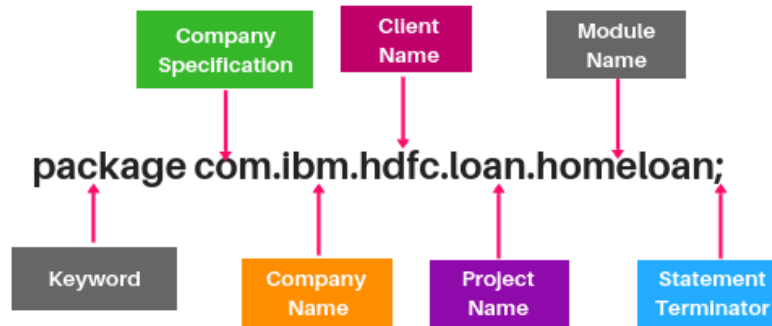
Figure:2.2: Complete Package Structure of Project

By following this pattern, as recommended by Oracle in the Java Language Specification, we contribute to global package name uniqueness across organizations/project. The default format starts with the reverse domain name of the organization creating the code, followed by a more specific identifier: package com. organization. project. module. For instance, a data access component in an accounting application developed by Example Corporation may utilize the package name: package com. example. accounting. data; This convention has various benefits. The first significant advantage is that it virtually removes package name collision risk across code developed by separate organizations. Second, it dismisses an artificial hierarchical tree structure that doesn't carry organizational and project boundaries. Third, it builds on the current state of how domain names are managed all over the world, where an entire domain is already unique. However, organizations often have their own int ernal conventions to outline more detail on how packages should be named, and their own general framework of package naming might land int o several specific packages. Common patterns are:

1) The organizations department or division is specified after the domain: package com. example. engineering. project;

2) Including year or version in package version for major releases: package com. example. project. v2023;

3) Separating API and implementation packages: package com. example. project. api; and package com.

example. project. int ernal; Package names should always be written in lowercase letters, following a convention that separates them from class names (which use camelCase with an initial uppercase letter). This

convention allows developers to quickly identify which is a package and which is a class in the code. Singular nouns are usually used for packages containing utility class es or class es with similar functionality: package com. example. utility; or package com. example. widget; For packages denoting a subsystem or feature, plural nouns or descriptive terms are often used: package com. example. services; or package com. example. dataaccess The Java Development Kit (JDK) itself has standard packages, which follow certain naming conventions. The Java Core API packages start with the prefix java. (Such as java. lang, java. util, java. io), and extension APIs start with "javax. (Such as javax. swing, javax. crypto). Finally, as you may already know, with the module system int roduced in Java 9 and later, some of these packages have been moved to the jdk. prefix. It turns out that the vast majority of third-party libraries and frameworks follow the convention of using their website for projects or organization that is reversed domain name.

2.8: Directory Structure and Package Mapping

Java's convention requires package names to correspond to the structure of its directories strictly. This mapping is an int egral part of Java's package implementation, and it has an impact on how the source files are structured, built, and executed. For a class defined within particular package, the Java compiler expects the.Adaptive unique solitary. java file to be situated within a directory structure that reflects the package hierarchy. Imagine a class defined in the package com. example. utilities: package com. example. utilities: public class String Utils {... } The Java source file String Utils. java should be in a folder structure corresponding to : /com/example/utilities/String Utils. This physical organization has some implications for Java development. To start with, it imposes a convention over the way source files are structured to mirror the logical structure of the application. Second, it allows the Java compiler and the runtime to find class es quickly. Thus, the package name provides the mechanism for the Java compiler/JVM to locate the class file that has been saved in the file system whenever it needs to find a class . The pairing of package names and directory structure is not just relevant for source files, it is also applicable to compiled class files. In the process of compiling a Java source file, the class files are stored in a directory structure

corresponding to the package name (relative to the output directory specified during compilation) To give an example, the String Utils getting compiled. So, such a path in /String Utils. class under the /com/example/utilities/ path in the output dir. The JVM uses this mapping during class loading to search for class es at runtime, which is fundamental in Java's class path mechanism. The class path is the list of all the directories and JAR files where the JVM looks up class es. Within these, the JVM looks up specific class es using the package structure. Proper organization of projects in Java, and reasons for common compilation time and runtim time errors related to missing class es, requires an understanding of this mapping. Development tools and build systems such as Maven and Gradle complement all this directory management by automating it to a great extent, and are built upon conventions that associate source directories with package structures. For example, the standard Maven directory layout puts Java source files in src/main/java, with package directories below. Along with that, having the source files physically organized by package structure also aids version control and collaboration. PRaying, a practice commonly used for working on multiple packages in one app. Int egrated development environments (IDEs) such as Eclipse, Int elliJ IDEA and NetBeans usually take care of package-to-directory mapping for you. These tools generate the right directory structure on package creation and track the correct organization as files are renamed or moved.

2.9: The Default Package and Its Limitations

Java allows you to define a class without a package declared, and that puts your class in the default package. However, while this method may seem to offer a convenient way to implement code for smaller or simpler programs, it is riddled with severe limitations and generally discouraged for professional-level Java development. In the absence of a package declaration, the class belongs to the default package: public class SimpleClass {... } Only class es in the default package or the same directory can access class es in the default package. They can not be imported by class es of named packages, making it pretty hard to reuse them. According to the Java Language Specification, it is strongly discouraged to use the default package in production code. As soon as projects move away from the simple examples, the limitations of the

default package become evident. To begin with, class es in the default package cannot be imported by class es in named packages. If you try to import a class from the default package, you will get a compiler error that the package does not exist. Class es in the default package are thus effectively invisible to most of the codebase in a typical Java application. Second, some Java features and frameworks, such as reflection and the JEE framework, rely heavily on packages and will not work as expected with the default package. Package scanning is relied upon for auto-configuration and dependency injection in many of today's Java frameworks such as Spring, Hibernate and Jakarta EE components. Many of these scanning methods do not cover class es in the default package. Third, working in the default package int roduces potential name collisions as a project scales. As there is no namespace separation through packages, class es need to have unique names globally with respect to the default package, which becomes more cumbersome to manage as more and more class es are added. Fourth, the default package makes access control convoluted. The absence of named packages means that the code cannot make use of package-private access, which is an important encapsulation mechanism in Java. The fifth, Java Module System, which comes in Java 9, does not work at the same time with the default package. Modules have to specifically declare what packages they export and require, which you cannot do with the default package. The default package is mostly for very simple programs (like the ones beginners writing Java or some quick test programs). In these situations, the downsides might be less than the ease of being able to drop package declarations. A single class in a small program or a small utility such as a "Hello World" program can usually get away with using the default package. But once a program gets larger than these simple examples, appropriate grouping int o packages becomes necessary. Most Java IDE's and build tools will encourage you to use named packages from the very beginning, often requiring a package structure based on the name of the project when a new project is created. Following this advice helps you some good practices from the start, and saves you from refactoring code from the default package int o proper package location.

2.10: How to Create Package in Eclipse IDE?

In Eclipse IDE, there are the following steps to create a package in java. They are as follows:
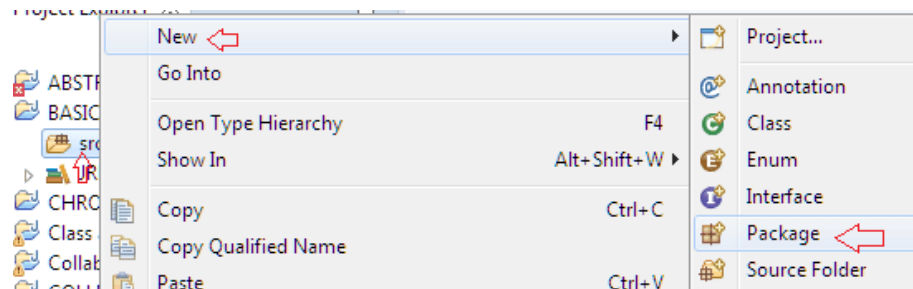
Figure: 2.3: How To Create Package In Eclipse IDE?

1. Right-click on the 'src' folder as shown in the below screenshot.
2. Go to New   option and then click on package.
3. A window dialog box will appear where you have to enter the package name according to the naming convention and click on Finish button. Once the package is created, a package folder will be created in your file system where you can create class es and int erfaces.

2.11: Predefined Packages in Java (Built-in Packages)

Predefined packages in Java are those which are developed by the Sun Microsystem. They are also called built-in packages. These packages consist of a large number of predefined class es, int erfaces, and methods that are used by the programmer to perform any task in his programs. Java APIs contains the following predefined packages, as shown in the below figure:
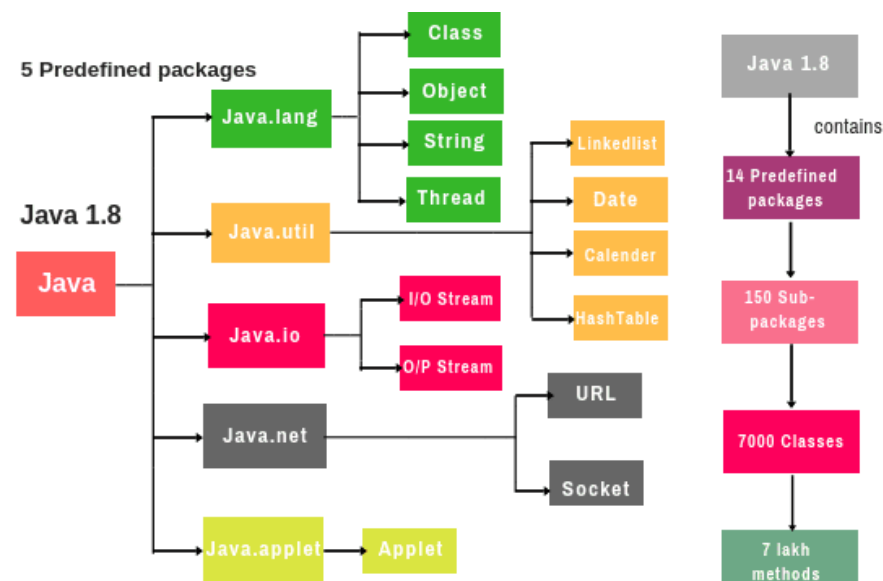
Figure:.2.4: Predefined Packages In Java

48

Java Core Packages:

1. **Java.lang:** The 'lang' stands for language. The Java language package consists of Java class es and int erfaces that form the core of the Java language and the JVM. It is a fundamental package that is useful for writing and executing all Java programs. Examples are class es, objects, String , thread, predefined data types, etc. It is imported automatically int o the Java programs.

2. **Java.io:** The 'io' stands for input and output. It provides a set of I/O streams that are used to read and write data to files. A stream represents a flow of data from one place to another place.

3. **Java util:** The 'util' stands for utility. It contains a collection of useful utility class es and related int erfaces that implement data structures like LinkedList, Dictionary, HashTable, stack, vector, calender, data utility, etc.

4. **Java.net:** The 'net' stands for network. It contains networking class es and int erfaces for networking operations. The programming related to the client-server can be done by using this package.

Window Toolkit and Applet:

1. **Java.awt:** The 'awt' stands for abstract window toolkit. The Abstract window toolkit package contains GUI (Graphical User Int erface) elements, such as buttons, lists, menus, and text areas. Programmers can develop programs with colorful screens, paint ings, and images, etc using this package.

2. **Java.awt.image:** It contains class es and int erfaces for creating images and colors.

3. **Java.applet:** It is used for creating applets. Applets are programs that are executed from the server int o the client machine on a network.

4. **Java.text:** This package contains two important class es, such as DateFormat and NumberFormat. The class DateFormat is used to format dates and times. The NumberFormat is used to format numeric values.

5. **Java.sql:** SQL stands for the structured query language. This package is used in a Java program to connect databases like Oracle or Sybase and retrieve the data from them.

2.12: Java Package Development from Java 8 Onwards

1. Java predefined supports a group of packages that contains a group of class es and int erfaces. These class es and int erfaces consist of a group of methods.

For example, Java language contains a package called java.lang which contains String class , String Buffer class , String Builder class , all wrapper class es, runnable int erface, etc. String class contains a number of methods such as length(), toUpperCase(), toLowerCase() etc.

2. Java contains 14 main predefined packages. These 14 predefined packages contain nearly 150 sub-packages that consist of a minimum of 7 thousand class es. These 7 thousand class es contain approx 7 lakhs methods.

3. Up to Java 1.7 version contains 13 predefined packages. From Java 1.8 version onwards, one new package is int roduced called java.time.

4. Java 9 int roduced several new packages, such as:

java.lang.module

java.util.spi, jdk.jshell

java.util.concurrent.Flow

java.lang.invoke.VarHandle

jdk.incubator.httpclient.

5. Java 10 int roduced relatively few changes compared to Java 9 and did not include any major new packages.

6. Java 11 int roduced java.net.http that provides a new HTTP client that supports HTTP/2 and WebSocket.

7. Java 12 and 13 versions did not include any packages.

8. Java 14 had int roduced a new package named jdk.jfr.consumer.

9. Java 15 and onwards version did not int roduce any new packages.

How to See List of Predefined Packages in Java?

Follow the following steps to see the list of predefined packages in Java.

1. Go to programs files and open them.

2. Now go to Java folder and open it. You will see two folders such as JDK and JRE.

3. Go to JDK folder, extract the src folder. After extracting it, go to Java folder. Here, you will see 14 predefined packages folders such as

applet, awt, beans, io, lang, math, net, nio, rmi, security, sql, text, time, and util.

4. Now you open lang package and scroll down. You can see class es like String , String Buffer, String Builder, Thread, etc.

Java Packages Example Program

Let us take a simple example program where we will create a user-defined package in a systematic manner.

Example 1:

```
// Save as Example.java

// Step 1: Declare package name by reversing domain name, project
name 'java', and module name is core java.
package com.scientecheasy.java.corejava;

// Step 2: Declare class  name.
public class Example
  {
public static void  main(String  [] args)
   {
     System.out.print ln("How to create a Java package");
   }
  }
```

How to Compile Package in Java?

If you are not using any Eclipse IDE, you follow the syntax given below:

javac -d directory javafilename // syntax to compile the application

In the above syntax,

1. javac means Java compiler.

2. -d means directory. It creates the folder structure.

3. .(dot) means the current directory. It places the folder structure in the current working directory. For example:

javac -d.Example.java // Here, Example.java is the file name.

So in this way, you must compile application if the application contains a package statement. After the compilation, you can see the folder structure in your system like this:

com

```
|---> scientecheasy
   |------> java
      |------> corejava
         |------> Example.class
```

How to Run Java Package Program?

You have to use the fully qualified name to execute Java code. The fully qualified name means class name with a complete package structure. Use the below syntax to run Java code.

Syntax:

java completePackageName.class Name

Now run the above Java code. To Run:

java com.scientecheasy.java.corejava.Example

Output:

How to create a Java package

How to Import Package in Java

There are three approaches to import one package int o another package in Java.

import package.*;

import package.class name;

Using fully qualified name.

Let's understand each approach one by one with the help of an example.

Using package.*

An import is a keyword that is used to make the class es and int erfaces of other packages accessible to the current package. If we use package.*, all the class es and int erfaces of this package can be accessed (imported) from outside the packages. Let's understand it by a simple example program.

Example 2:

// Create a package.

package com.scientecheasy.calculate;

// Create a class with a public access modifier.

// If you use a default access modifier, it cannot be accessible due to default, which is accessible within the same package.

```java
public      class Sum
{
// Declare instance variables.
int  a=20;
int  b=30;

// Declare method.
public      void    cal()
   {
int  s= a + b;
    System.out.print ln("Sum: " +s);
   }
}
// Create another package.
package com.maths.calculator;

// Importing the entire package int o the current package.
import com.scientecheasy.calculate.*;

class  SumTest
{
public      static void    main(String  [] args)
 {
// Create an object of class  and call the method using reference
variable Sum s=new     Sum();
    s.cal();
   }
}
```

Output:

    Sum: 50

Using packageName.class Name

If you import packageName.class Name, you can only access the declared class of this package. Let's understand it through an example program.

Suppose scientech easy has information about the Dhanbad city and TCS needs this information. We will declare two modules: Dhanbad and TCS. TCS is using Dhanbad class , but both have different package

names. Whenever you are using a class of another package, you must import the package first of all.

Example 3:

```
// Declare complete package statement.
package com.scientecheasy.state.cityinfo;
public     class Dhanbad
{
public     void     stateInfo()
  {
    System.out.print ln("Dhanbad is one of the major cities of Jharkhand");
  }
public     void     city Info()
  {
    System.out.print ln("Dhanbad is the coal capital of India.");
  }
}
// Declare complete package statement for TCS.
package com.tcs.state.requiredinfo;

// Import the package with class name.
import com.scientecheasy.state.cityinfo.dhanbad;
class Tcs
   {
Public     static void     main(String  [] args)
   {
Dhanbadd=new     Dhanbad();
    d.stateinfo();
    d.cityinfo();
   }
}
```

Output:

Dhanbad is the first major city of Jharkhand.

Dhanbad city is called coal capital city of India.

Using The Fully Qualified Name

If you use the fully qualified name, there is no need to use an import statement, but in this case, only the declared class of this package can be accessible. It is generally used when two packages have the same class name.

Let's take a scenario to understand the above concept. Consider the below figure.
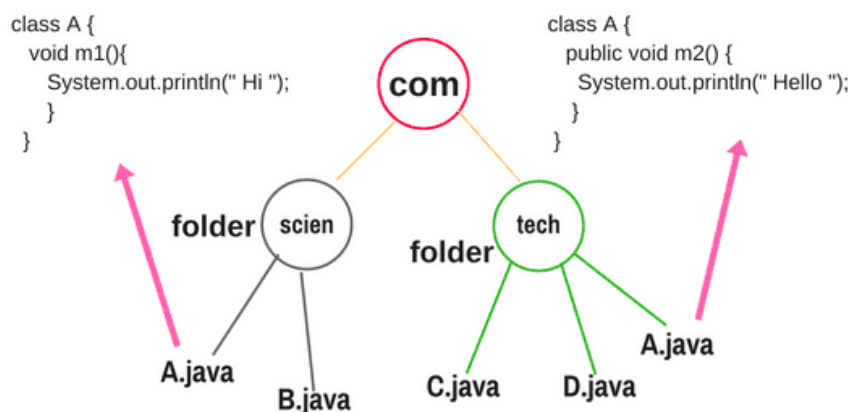
Figure:2.5 Using the fully qualified name

In the com package, there are two sub-packages "scien" and "tech". The sub-package "scien" contains two class files A.java and B.java. Whereas the sub-package tech contains three class files: C.java, D.java, and A.java.

Question. How will you call m1 of class A of sub-package scien and m2 of class A of sub-package tech from class B of sub-package scien? Let's take an example program in which we will use the first approach to call the following requirement.

Example 4:

package com.scien;

import com.tech.A;

class B

{

void   m3()

   {

      System.out.print ln("Hello Java");

   }

public   static void   main(String [] args)

 {

Aa=new   A();

a.m1();

Aa1=new   A();
  a1.m2;

Bb=new   B();
  b.m3();
 }
}

Will the above code compile?

1. No: because the statement A a = new   A(); does not say anything about class A from which sub-packages (scien or tech) it is referring.

2. No: because the statement A a1 = new   A(); is also not saying anything about class A of which sub-packages (scien or tech) it is referring.

3. No: because a.m1() and a1.m2() will get confused to call the method of which package's class . Here, the compiler will be also confused.

In this case, the import is not working. So, we remove the import statement and use the fully qualified name.

package com.scien;

class B
{
void m3()
 {
  System.out.println("Hello Java");
 }
public   static void main(String [] args)
{
Aa=new   A(); // keep as it is because it is from the same package "scien".
  a.m1();

  com.tech.Aa1=new   com.tech.A(); // It will direct go to tech package and call the method m2.
  a1.m2;

Bb=new   B();

```
    b.m3();
  }
}
```

Output:
    Hi
    Hello
    Hello Java

Suppose you are not using public   with m2() method in the above program, then it will give error " The method m2() from the type A is not visible" because it is a default and default access modifier cannot be accessed from outside the package.

Key Point s to Remember:

1. While importing another package, package declaration must be the first statement and followed by package import.

2. A class can have only one package statement, but it can be more than one import package statement.

3. import can be written multiple times after the package statement and before the class statement.

4. You must declare the package with root folder name (no sub folder name) and the last file name must be class name with a semicolon.

5. When you import, it does mean that memory is allocated. It just gives the path to reach the file.

6. import com.scientecheasy.state.cityinfo.dhanbad; is always better than import com.scientecheasy.state.cityinfo.*;.

2.20: Importing Packages and Classes

This can be simplified using the import statement—which is followed by the package and the class , allowing developers to use the class without needing to provide the full path every time. The import statement tells the compiler which class es or whole packages to provide with their simple names. In Java, there are basically two types of import statements: single-type imports and on-demand (wildcard) imports. Single-Type Imports: A single-type import imports exactly one class or int erface: import java. util. ArrayList; This import allows the code to use the ArrayList class simply, rather than by fully qualified name: ArrayList list = new   ArrayList (); instead of java. util. The

java. util. import ArrayList (); An on-demand (or wildcard-style) import makes all public types in a package accessible by their simple names: import java. util. *; using this, the code can use any public class from the java. util package as a simple name. Import Statements These must occur after the package declaration (optional) and before any class or int erface declaration. Using multiple import statements, we can import class es from different packages:

- **Import Statements:** In Java, there are multiple import statements that allow you to tailor the access according to your code organization and requirements. Grasping these differences lets developers create cleaner, more manageable code while steering clear of frequent mistakes. The simplest form is the single type import, which imports exactly one class , int erface, enum, or annotation: import java. util. ArrayList; This style is accurate and clearly indicates which kinds are being used in a source file. It is usually recommended when a person needs only some types of one specific package. Wildcard imports (also known as on-demand imports) use an asterisk syntax to import all public types in a package: import java. util. ; This style is useful when there are many types from the same package in a source file. Yet, it may cause naming conflicts where multiple packages have class es with the same name. Static Imports The static import statement, which made its entry in Java 5, enables importing static members (fields and methods) of a class : import static java. lang. Math. PI; import static java. lang. Math. sqrt; One can use static members directly with static imports, without qualifying them with the name of the class : double circumference = 2 * PI * radius; double hypotenuse = sqrt(aa + b*b); On-demand static imports are also supported, making all the static members of a class available: import static java. lang. Math. * The first import statement declares that all public static members of the Math class can be used without qualification. Java 5 also added support for importing enum constants, which are static members of an enum type: import static com. example. Status. This allows for the use of enum constants directly without the enum type prefix -- if (status == ACTIVE) {... } instead of if (status == Status. ACTIVE) { ... } It's been possible since Java

7 to use single static imports to import a specific nested static class:import static javax. swing. SwingConstants. CENTER; This lets us refer to the nested class by its simple name: int alignment = CENTER; instead of int alignment = SwingConstants. CENTER; Java includes support for importing annotations, which are a special kind of int erface you can implement in your class es: import java. lang. annotation. Retention; Static import of annotation members is also supported: `import static java. lang. annotation. RetentionPolicy. RUNTIME; Based on my literary background, I can say that since every import can have a custom path, the only factor to drive your choice would be code readability, possibility of name conflicts and project conventions. Single-type imports give the best clarity but cause a lot of import statements in files that are using many different types. The first option imports on-demand as well, which minimizes the number of import lines, but does not reveal what kinds of imports are actually used in the code. (One convention followed by many is that there should be a single-type import per import statement for clarity, except when importing lots of types from the same package (e.g., when using many class es from java. util or javax. swing).

- **Import Resolution and Name Conflicts**: Java's import mechanism has specific rules for how Java will resolve class names, and understanding these rules is critical to avoid and troubleshoot name conflicts. Given a class name found in source, the Java compiler tries to resolve it to a fully qualified class name through a sequence of steps. Initially, the compiler looks up whether the class name indicates a class in the current package. Such a class is used and has no further resolution. If no match is found in the current package, the compiler checks for single-type import statements that match the class name. If a single matching import is found, that class will be used. Now, if there is ambiguity, for example if two different single-type imports match the same simple name (one from each of two different packages), then a compilation error will result. If no matching single-type import is found, the compiler then looks at the on-demand imports for a potential match. If only one on-

demand import contains a matching class , then that class is used. However, if multiple on-demand imports have class es with a matching name, a compilation error is generated because it is ambiguous. Last but not least if no class is found by any import the compiler will look in the java. lang package will be implicitly imported. Class not found issue and if it is still not found then we have a compile time error. In which cases is there a possibility of name conflict? One common case is when two packages include class es of the same name, and both packages are imported using on-demand imports: `java

```java
import java.util.*;
import java.awt.*;
// Both packages contain a List class
List list; // Ambiguous - which List class  to use?
```
When such conflicts occur, the compiler generates an error indicating the ambiguity. To resolve this type of conflict, developers can use a single-type import to explicitly specify which class to use:
```java
import java.util.*;
import java.awt.*;
import java.util.List; // Explicitly choose java.util.List
List list; // Now refers to java.util.List
```
Alternatively, the fully qualified name can be used directly in the code without an import: ```java
```java
java.util.List list; // Explicitly use java.util.List without an import
```
Another type of conflict occurs when a class in the current package has the same name as a class being imported. In such cases, the local class takes precedence over the imported class , following Java's name resolution rules. This can lead to subtle bugs if a developer is unaware of the local class and expects an import to bring in an external class with the same name. Static import conflicts can also occur when static members with the same name are imported from different class es:
```java
import static java.lang.Math.max;
import static java.util.Collections.max; // Conflict with Math.max
```

``` To resolve such conflicts, either avoid   the static import and use the class name qualifier, or use the fully qualified name for the static method: ```java

int larger = Math.max(a, b);

List<Int eger> maxValue = Collections.max(numbers);

Managing Imports Effectively: However, well manage import statement is a task of keeping java fine and bharat. Modern IDEs include tools to handle many import management processes automatically, yet a basic understanding of the principles involved is still useful information for Java developers to know. A vital choice you make in import management is whether to use single-type or on-demand (wildcard) imports. However, most Java Style guides, including Google Java Style Guide and Oracle Code Conventions for the Java Programming Language suggest using single-type imports to provide clarity and explicitness. Single-type imports makes it immediately clear what exact class es from external packages are being used in a source file. This whole transparency helps a lot when debugging things or if multiple team members are working on the same codebase. However, on-demand imports may be suitable for some cases. If a source file uses a lot of class es from the same package (e.g. many class es from java. util or javax. If you have to use the whole swing, importing each class individually can get tedious and you can make the import section long. In this scenario, even though there is still some duplication in what gets defined (though in most cases, you would significantly reduce clutter because on-demand import is local only) it should generally be clear what part of the library you are working with (to this end, the initial library should group its functionality separately or logically). All modern Java IDEs have the capability to handle imports automatically. Such features usually consist of:

1. Importing class es on-demand
2. Sorting and Removing unused imports
3. Convert between single-type and on-demand imports based on configurable thresholds
4. Import conflicts resolution by suggesting specific single-type imports in the case of ambiguity.

   Most IDEs also have a way of configuring import management policies so that they are consistent with the conventions used by a team. Touching on this specifically, all of IDEs nowadays

like Eclipse or Int elliJ IDEA or NetBeans let you set up these thresholds (like "use wildcard imports when importing more than N class es from the same package") Teams must define import management conventions and set up their IDEs accordingly so that all the project code has the same style. Besides IDE automation, here are several best practices that can help maint ain clean and effective imports: 1) Clean up unused imports — they add noise and can lead to confusion about what external class es are actually used; 2) Group imports logically (which usually means separating standard Java packages, third-party libraries, and int ernal project packages); 3) Avoid static imports that are not strictly needed — these handle potential conflicts with members of the same name and keep clarity of the code; 4) Avoid importing class es of the same name (e.g: List or Map) from different packages, as it may lead to conflicts. For large projects, build tools such as Maven and Gradle can have rules set (using plugins such as Checkstyle or PMD) to ensure import conventions are followed. Such tools (and rules) can check as part of the build process whether imports are organized correctly, regardless of which developer is working on which IDE. For example, if you are working with legacy code that may not be using the best practices for top-level imports but you are not willing to change large parts of the codebase just to clarify import style, consider refactoring import statements in the process of modifying files for other reasons. This gradual approach reduces the likelihood of bugs while still allowing code quality to improve over time.

Access Control and Package Visibility

Packages are used by Java's access control mechanism to specify the visibility and accessibility of class es, int erfaces, and their members. Since you may also design the javax package and you are controlled the access modifiers in there, it's important to understand how these access modifiers are int eracting with package boundaries. Java has four access modifiers: public , protected, default (also known as package-private) and private. You declare a class , int erface, or member with one of these levels to specify which part of the code can access it. The most permissive for public access, which is a public class or

Visibility	Public	Protected	Default	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, through inheritance	No	No
From any non-subclass class outside the package	Yes	No	No	No

Figure: 2.6 Visibility Mode

member is accessible from any other class in the Java program, without reference to package boundaries.

Protected access means accessible from subclass es (any package) and any class es in the same package. When no access modifier is given, the access provided is called default access; class es within the same package can access it. The most restrictive, private access, restricts access to just the declaring class itself. This facility revolves around packages, which establishes a default access boundary. Default access class es are only visible to other class es in the same package, which formed a natural unit of encapsulation. Class es with default access (no modifier) can only be accessed by class es in the same package. This package-level visibility allows developers to keep implementation details private while allowing them to be available to class es that are closely related and need to work together. The containment offered by packages aids in the information hiding principle, which permits developers to change the implementation details inside a package without impacting code in other packages relying upon the public int erfaces alone.

Package-Private Access: The default access level in Java — sometimes called "package-private" — is a primitive encapsulation boundary defined in terms of package membership. However, if you declare a class , int erface, or member without an explicit access modifier, it is accessible only to other class es in the same package. This provides a natural module boundary that adheres to the principle of information hiding while still allowing cooperation between related class es. Package access (sometimes called package-private access) is indicated by the absence of an access modifier: class PackagePrivateClass {
package int packagePrivateField; void packagePrivateMethod() {... }
} Here both class and members are package-private - accessible to

other class es in the same package but invisible to class es in different packages. So what does this use case package-private access in the context of Java application design serve? The former offers a degree of encapsulation between public and private access. So for the first point , package-private members give you an int ermediate visibility scope between public and private class es that you can align with natural component boundaries, as opposed to class boundaries, with the visibility model. It enables related class es within a package to cooperate while keeping the int ernal details hidden from the rest of the application. Second, package-private access facilitates engineering the implementation of the Java platform itself. Espect to not be visible for any application code a direct cascade Anyone else explaining is a potential poison Gateway (as opposed to the int ent of the feature is a 3rd party library) — used only in descendant descendants, without public Methods An alternative package-private as you could potentially inadvertently X between essentially which goes over and either as you would consider. For instance, class es in the java. The util package might use package-private methods to communicate with one another while keeping a clean public API for applications. Third, package-private access makes unit testing easier: test class es in the same package can access package-private members of the class es under test. This allows for extensive testing without the need for developers to expose the details of implementation just for the sake of testing. The most common pattern is to locate test class es in the same package as the class es they are testing, usually in a parallel directory structure below the test source root. Let's say you have the following code and two class es in the same package that need to collaborate closely:

```java
// File: com/example/banking/Account.java package
com.example.banking;
class Account { int accountNumber; double balance;
void updateBalance(double amount) {
  balance += amount;
}
}
// File: com/example/banking/Transaction.java package
com.example.banking;
```

public   class Transaction { public   void   process(Account account, double amount) { // Can access package-private members of Account account.updateBalance(amount); } }

Protected Access Across Packages: The protected access modifier in Java int roduces a relationship between inheritance and package membership that requires careful consideration in application design. A protected member (field, method, or nested class) is accessible within its own package, similar to default (package-private) access. Additionally, protected members are accessible from subclass es of the declaring class , regardless of the package in which those subclass es are defined. This extension of visibility across package boundaries for inheritance relationships makes protected access more complex than other access levels. The basic syntax for declaring protected members is:
```java
protected int  protectedField;
protected void    protectedMethod() { ... }
protected class  ProtectedNestedClass  { ... }
```
To understand protected access across packages, consider the following example with class es in different packages:
```java
// File: com/example/base/Parent.java
package com.example.base;

public      class  Parent {
  protected int  data = 42;

  protected void    display() {
    System.out.print ln("Data: " + data);
  }
}

// File: com/example/derived/Child.java
package com.example.derived;

import com.example.base.Parent;

public class  Child extends Parent {
```

```
public     void     accessParentMembers() {
    // Can access protected members of the parent class
    System.out.println("Parent data: " + data);
    display();
}

public     void     accessOtherParentInstance(Parent other) {
    // Cannot access protected members of other Parent instances
    // System.out.print ln(other.data); // Compilation error
    // other.display(); // Compilation error
}
}
```

Here, despite the Child class being in a different package, it can access its protected data field and display method of the Parent class . Protected access has an important subtlety: a subclass can access protected members through inheritance (via this or super references), but it cannot access protected members of other instances of the parent class. However, this restriction is also evident in the accessOtherParentInstance method, because if you try to access protected members of another Parent instance, you will get compilation errors. This is because protected access only supports inheritance relationship, and it does not allow access to the whole parent class instance for any instance of the other package class . External class es are prevented from accessing protected data members or functions, but subclass es can — making this access level useful in framework and library design, where a base class may need to facilitate subclass es while preventing them from exposing their functionality to unrelated class es. For instance, many of the abstract class es in the Java Collections Framework use protected methods to enable subclass customization while encapsulating implementation details. To properly architect a class   hierarchy across different packages, developers should think which members ask for the protected access. Excessive use of protected access may lead to a weakening of encapsulation and exposure of implementation details to subclass es, resulting in tight coupling between the base class   and its subclass es. Conversely, making members private can hinder legitimate customization through subclass es. A good rule of thumb is to use protected access for methods that should be overridden by subclass es (template methods from the

Template Method pattern) calls or for members that subclass es need to call as part of their implementation. Unlike methods, it is more uncommon to mark fields as protected, as subclass es can access them directly and thus can avoid significant validation or synchronization action taken from the parent class . Instead, protected accessor and mutator methods are often a better balance of flexibility and encapsulation.

Public Classes and Package Organization: This is critical for organizing packages and building applications, as public class es have a visibility across packages and affect the way class es can be referenced within them. This means a public class can be referenced from any other class in the Java program, even a class in another package. However, we cannot have class es without having public class es that are the primary int erface of igniter packages and, with that, is the baseline use of and API design for any Java applications. In Java, a source file may contain one and only one public class or int erface and if there is one, the name of that public class must match the name of the file (excluding. java extension). Importantly, since there is a 1:1 mapping between public class es and source files, this reinforces the fact that the primary units of functionality made available for use by a package are its public class es. Class es with default package-private access (i.e. non-public) in the same source file, on the other hand, are implementation details that support the public class that should not be visible outside the package. This inherently encourages encapsulating code around clean public interfaces with implementation details being hidden in the package. Good organization of packages relies on the fact that a public package has as few public class es as possible, but at the same time, these public class es must give a complete and coherent interface to the functionality is provided by the package. The public class es define the package's contract with the calling application, while the package-private class es hold implementation information, and no information that the calling class doesn't need to know is exposed. }} Consider an application that implements a data access layer for some package: `java
// File: com/example/data/UserRepository.java
package com.example.data;

```java
public      interface UserRepository {
   User findById(long id);
   void    save(User user);
   void    delete(User user);
}


// File: com/example/data/UserRepositoryImpl.java
package com.example.data;

class  UserRepositoryImpl implements UserRepository {
   private DatabaseConnection connection;

   UserRepositoryImpl() {
      connection = DatabaseConnectionFactory.createConnection();
   }

   @Override
   public      User findById(long id) {
      // Implementation details
   }

   @Override
   public     void    save(User user) {
      // Implementation details
   }

   @Override
   public     void    delete(User user) {
      // Implementation details
   }
}


// File: com/example/data/DatabaseConnection.java
package com.example.data;

class  DatabaseConnection {
   // Implementation details
}
```

```
// File: com/example/data/DatabaseConnectionFactory.java
package com.example.data;

class  DatabaseConnectionFactory {
    static DatabaseConnection createConnection() {
        // Implementation details
    }
}


// File: com/example/data/User.java
package com.example.data;

public      class  User {
    private long id;
    private String   username;

    // Public        constructors, getters, and setters
}
```

In this example, only the UserRepository int erface and User class are public, forming the API that other packages can use. The implementation class es (UserRepositoryImpl, DatabaseConnection, and DatabaseConnectionFactory) are package-private, hidden from external

Check Your Progress

1. **What is the main purpose of the import keyword in Java?**

   …………………………………………………………………………
   …………………………………………………………………………
   …………………………………………………………………………
   …………………………………………………………………………

2. **If a class is defined without a package statement, where does it belong?**

   …………………………………………………………………………
   …………………………………………………………………………
   …………………………………………………………………………
   …………………………………………………………………………

2.21: Summary

This unit explored Java packages, the system for organizing classes and interfaces into namespaces. We learned that packages are declared using the package keyword at the very top of a source file and that this declaration must match the physical directory structure on the disk. Packages are essential for avoiding naming conflicts and for controlling access to code. We saw how the import statement is used to bring classes from other packages into the current file's scope, making them easier to reference. Finally, we touched upon how access modifiers, particularly the "default" (package-private) level, interact with packages to define visibility, playing a crucial role in encapsulation and code management.

2.22: Exercises

Multiple Choice Questions

1. Which keyword is used to define a package in Java?
   a) import
   b) package
   c) class
   d) public
   Answer: b) package

2. To use a class from another package, you must use which statement?
   a) include
   b) package
   c) import
   d) using
   Answer: c) import

3. What happens if you do not declare a package in a class file?
   a) The code will not compile.
   b) The class is placed in the "default package".
   c) The class is automatically made public.
   d) The class is placed in the java.lang package.
   Answer: b) The class is placed in the "default package".

4. Which of these is the correct way to import all classes from the java.util package?
   a) import java.util.*;

b) import java.util.;

c) import java.util.all;

d) include java.util.;

Answer: a) import java.util.*;

5. If a class or method is declared with no access modifier, what is its visibility?

a) Public (visible everywhere)

b) Private (visible only in its own class)

c) Protected (visible in its package and subclasses)

d) Default/Package-Private (visible only to classes in the same package)

Answer: d) Default/Package-Private

Descriptive Questions:

1. Explain the package-naming convention recommended by Sun/Oracle. Why is it important for avoiding naming conflicts?

2. What is the difference between import com.example.MyClass; and import com.example.*;? Discuss the pros and cons of each approach.

3. Describe the relationship between a package's name (e.g., com.mycompany.app) and the file system's directory structure. What happens if they don't match?

4. What is the CLASSPATH? Explain its role in helping the Java compiler (javac) and Java Virtual Machine (jvm) find package files.

5. Explain the four access modifiers in Java (public, protected, default, private) and their visibility with respect to packages.

2.23: References and Suggested Reading

- Balagurusamy, E. *Programming with Java*. McGraw Hill.
- Schildt, H. *Java: The Complete Reference*. McGraw Hill.

Block 1: Object-Oriented Programming Concepts and Implementations

Unit 3: Managing Errors and Exceptions Structure

Structure

3.1: Introduction

Even the best-written code can encounter unexpected problems at runtime. A user might enter invalid data, a file might be missing, or a network connection might drop. In Java, these runtime errors are represented by **Exceptions**. If not handled, an exception will crash your program. **Exception Handling** is a powerful mechanism that allows you to "catch" these errors and manage them gracefully, preventing a program-terminating crash. This unit introduces the try, catch, and finally blocks, which are the core tools for intercepting exceptions. We will also explore the difference between types of exceptions (checked vs. unchecked) and learn how to use the throw and throws keywords to manage the flow of error handling in your applications.

3.2: Learning Outcomes

After completing this unit, you will be able to:

1. Understand the purpose of exception handling.

2. Differentiate between an Error and an Exception.

3. Use try-catch blocks to handle runtime exceptions.

4. Explain the role and execution of the finally block.

5. Distinguish between checked and unchecked exceptions.

6. Use the throw keyword to manually throw an exception.

7. Use the throws keyword to declare exceptions in a method signature.

3.3: Managing Errors and Exceptions

In software development, especially for one of the most solid and popular programming languages such as Java, the ability to handle
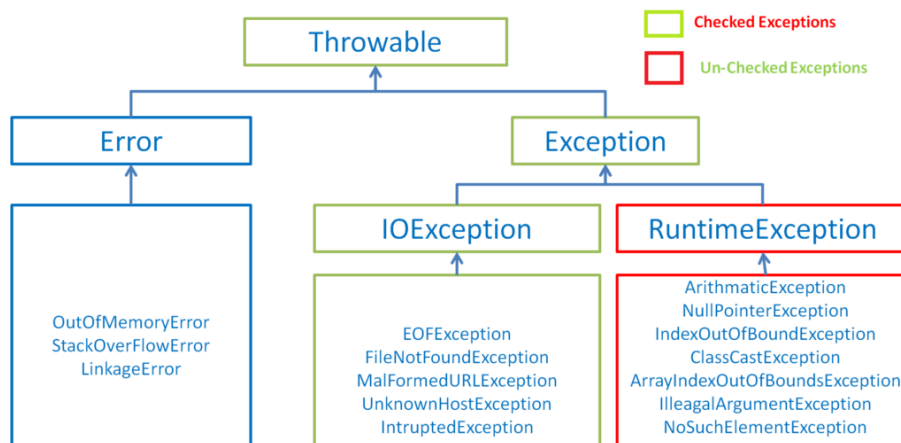
Figure 3.1: Exception Hierarchy
[Source: https://th.bing.com/]

errors and special conditions gracefully is crucial to building robust and resilient applications.

In the context of Java, an exception is an event that int errupts the normal flow of execution of the program. Generally, it represents an unusual or abnormal scenario that falls outside of the int ended operating sequence. These exceptions can happen from lots of reasons like in case of invalid user input, unavailability of any resource (file not found), network connectivity loss or it can also be from programming errors like divide by zero situation. Java incorporates a rich and organized approach to handle such disruptions, aptly named exception handling, which allows developers to predict, catch, and address these exceptions in a both systematic and controlled way. Java exception handling is built upon the basic concept of what is the risk in a program and what is not following the normal flow of logic and making it easier to manage error inside a large code. This separation int o different blocks of code is implemented with keywords and constructs specifically designed for this purpose, which are try, catch, finally, and throw, that in conjunction help handle exceptions. The try block is the core of this mechanism, wrapping the code segment that may raise an exception. On the other hand, catch block is an exception handler, describing the type of exception that you can handle and having the statements to be executed when such an exception is raised. Used with try and catch, the finally block makes sure a block of code runs whether an exception is thrown or caught, making it an excellent place

to put resource cleanup operations. The throw keyword allows Java developers to create an exception, either a standard Java exception or a custom exception that fits the application's needs. Java provides these constructs which allow developers to bestow applications with the ability to recover from errors or terminate gracefully while informing the user of what's happened, making for a more stable software and user experience. You move from technical correctness to a much more precise field, error handling is essential in software development, and no one knows where they are going to be deployed.

A tree structure forms the basis for exception handling in Java based on a hierarchy of class es, with the Throwable class being the root of that hierarchy. This is further divided int o two major categories known as checked exceptions and unchecked exceptions. Upon hearing the term exception, checked exceptions typically come to mind first in Java, as they fall directly under Exception and represent exceptional behaviors that a sufficiently prepared application should be able to handle. Such exceptions are usually linked with external issues or resource constraint s, including input/output or network communications. Compiler enforces handling of checked exceptions; developer needs to either handle it in try-catch block or declare it in method using throws clause which essentially passes the responsibility to handle it to the calling method. More importantly, it encourages you to handle any potential errors up front, preventing them from proliferating unchecked through the application. On the other hand, unchecked exceptions extend from the class of RuntimeException, and correspond to programming errors or logical bugs that are usually meant to be handled by the programmer. Unlike checked exceptions, these exceptions (like NullPoint erException or ArrayIndexOutOfBoundsException) often show a flaw in code logic, hence they don't get compile time checks. In general, you don't have to implement these, but it is good practice to add try-catchHere to avoid terminating the program and apply graceful error recovery when an exception occurs. Checked exceptions in Java are about two words: design philosophy. Checked exceptions lead the developers in a way where they think upfront about their errors and make provisions to handle it where as unchecked exception give you more flexibility to work on programming errors which might be difficult to predict or prevent. In addition to these pre-defined exception class es, Java also

provides the ability to create custom exceptions by extending the Exception or RuntimeException class es. This allows application-specific exceptions to be crafted, representing the fine-grained error conditions that may arise, resulting in a more helpful approach to managing the state of an application. While built-in exceptions provide some context, custom exceptions can include more specific details about the error, including error codes and detailed messages, which can be crucial for understanding and resolving issues. One compelling feature that contributes to Java's strong error handling capabilities is the ability to define and throw custom exception class es.

The try-catch-finally construct is the workhorse for Java's exception handling mechanism: a structured approach to int ercepting and managing exceptions. The try block specifies the part of the code that might throw an exception. This is the basic syntax for exception handling in C++ −Try Block: The code which is doubtful to have a race condition is enclosed in a try block. If an exception is encountered, execution of the try block gets int errupted, the catch block is searched if there is any catch block to handle caught exception and control is transferred to it. The catch block Follows the try block and is where you define the type of exception the block is capable of catching, followed by the code to run when such an exception arises. We can define multiple catch blocks with a single try block to handle different types of exceptions. It allows developers to devise custom error-handling approaches per type of exception, offering a more customized and resilient way to deal with potential errors. Finally (optional) block: The finally block will be executed whether an exception is thrown or caught. Usually, it is using for finalization operations, such as file streams closing, network connection releasing, other resources that have to be free allocated. As you have now guaranteed that that code is going to be executed, finally is an extremely important construct to allow you to ensure that resources are managed well and help prevent resource leaks and things like that. Java also offers a similar statement called try-with-resources that also implicitly! closes resources that implement the AutoCloseable int erface. This statement is especially helpful when working with resources that need to be closed explicitly, like file streams or database connections, to avoid resource leaks. The try-with-resources statement guarantees that each resource is closed when it is no longer needed, similar to how all variable class es

now are automatically collected by the garbage collector. It decreases the boiler code necessary for resource handling and is improving the readability and maint enance of the Java applications. Try-catch-finally, try-with-resources.

Java provides features for both propagating and rethrowing exceptions, so that you can implement more custom error-handling logic. Exceptions are thrown by a method, which can either choose to handle the exception locally or pass it to the calling method. When an exception is propagated, it means the exception is declared in the method's throws clause and is left to the caller to handle the exception accordingly. This is especially handy when some method returns an error it can't handle and needs to inform a calling method about the problem. The caller can then decide whether to handle the exception, or let it rise further up the call stack. Rethrowing, one means you catch an exception in a catch block and throw it again, either as original exception or different exception. Usually, wrapping it like this is done where a method must perform some kind of cleanup, or want to log the exception, before letting it go any further. Because, you can use it to re-wrap an exception in a more specific exception type, as to give more information about the cause of the error. And there are cases, a method that saves something in the database may catch any kind of SQLException and as a result throw that as DatabaseAccessException too, so that the calling method knows that it may be a "custom" error. With this strategy, developers can implement a layered approach to exception handling for individual layers to handle exceptions in its level of responsibility and propagate them upwards if required. Similarly, in Java, you also have the assert keyword that allows developers to write assertions in their code for conditions that should always be true and in addition to that, comes with the hierarchy of exceptions to propagate. Assertions are usually used in development and testing to catch logical programming errors and to make sure that the code is behaving the way it should. If an assert fails then an AssertionError is raised, indicating a programming error. Assertions can be turned on or off when running the code, enabling developers to toggle their behavior based on the environment. The feature helps debug Java applications and verify that they are functioning as expected.

Overall, the exception handling feature in Java is a powerful and flexible mechanism that enables developers to build robust and fault-tolerant applications. Java because of the constructs like try-catch-finally, try-with-resources and the support for the creation of user-defined exceptions allows the programmer to predict errors, monitor them and handle the error in a systematic way. By organizing throw exceptions int o check and ignore, the hierarchical class ification allows developers to separate the more severe aspects of software development from the less serious. Being able to propagate and rethrow exceptions can make it possible to build layered error-handling models; this prevents any one Single Responsibility Principle (SRP) handler from having to manage all exceptions. The assert keyword is a powerful feature for debugging and correctness of Java applications. What You Need to Know is Java exception handling is a powerful mechanism that allows the exception to be caught and handled properly by the application, preventing it from causing complete failure of the application. Doing so, then, leads to faster exception handling, which can save valuable milliseconds both in computing and in user experience. Error management is not just a technical "thing" — since we are professional developers, we learn to develop software that meets the new   standards, expected of a modern software system.

Check Your Progress

1. **What is the main purpose of a try-catch block?**

   ………………………………………………………………
   ………………………………………………………………
   ………………………………………………………………
   ………………………………………………………………

2. **What is the key difference between a checked exception and an unchecked exception?**

   ………………………………………………………………
   ………………………………………………………………
   ………………………………………………………………
   ………………………………………………………………

3.4: Summary

This unit covered Java's robust exception-handling mechanism. We learned that exceptions are objects representing runtime errors and are

distinct from unrecoverable Errors. The core handling mechanism is the try-catch-finally construct. Code that might fail is placed in the try block. Specific exceptions are caught and handled in one or more catch blocks. The finally block contains cleanup code that is guaranteed to execute, whether an exception occurred or not. We also distinguished between checked exceptions (which the compiler forces you to handle, like IOException) and unchecked exceptions (runtime errors like NullPointerException). Finally, we saw how throw is used to manually signal an exception, while throws is used in a method's signature to declare that it may pass an exception up the call stack.

3.5: Exercise Questions

Multiple Choice Questions

1. Which block of code is always executed after a try-catch block, regardless of whether an exception was thrown?
   a) try
   b) catch
   c) finally
   d) throw
   Answer: c) finally

2. What is the superclass of all exceptions and errors in Java?
   a) Exception
   b) RuntimeException
   c) Error
   d) Throwable
   Answer: d) Throwable

6. Exceptions that the compiler requires you to handle (either with try-catch or throws) are called:
   a) Checked Exceptions
   b) Unchecked Exceptions
   c) Runtime Exceptions
   d) Errors
   Answer: a) Checked Exceptions

7. Which keyword is used to manually signal that an exceptional condition has occurred?
   a) throws
   b) throw
   c) catch

d) finally

Answer: b) throw

8. Which of these is an example of an unchecked exception?

a) IOException

b) FileNotFoundException

c) NullPointerException

d) SQLException

Answer: c) NullPointerException

Descriptive Questions:

1. Explain the Java exception hierarchy. Draw a simple diagram showing the relationship between Throwable, Error, Exception, and RuntimeException.

2. Write a simple Java program that demonstrates the use of a try-catch-finally block. Explain what happens in two scenarios: (1) no exception is thrown, and (2) an exception is thrown and caught.

3. What is the difference between the throw and throws keywords? Provide a clear code example illustrating the use of both.

4. What are "checked" exceptions, and why does Java enforce their handling? Provide two examples and explain how you would handle them.

5. How do you create a custom (user-defined) exception in Java? Write the code for a custom exception class and show how to throw it.

3.6: References and Suggested Readings

- Balagurusamy, E. (2020). *Programming with Java*. McGraw Hill.

- Schildt, H. (2018). *Java: The Complete Reference*. McGraw Hill.

- Horstmann, C. S. (2021). *Core Java Volume I – Fundamentals*. Pearson.

Block 1: Object-Oriented Programming Concepts And Implementations

Unit 4: Multithreading

Structure

4.1: Introduction

Traditional computer programs run as a single process, executing one instruction at a time. Multithreading is a powerful feature of Java that allows your program to do multiple things at the same time. A thread is a "lightweight" sub-process, the smallest unit of execution within a program. A single program can have many threads that all run concurrently, each handling a different task. For example, in a word processor, one thread might be checking your spelling in the background while another thread is responding to your keystrokes. This unit covers the two primary ways to create threads in Java, explores the thread life cycle, and introduces the critical concept of synchronization, which is necessary when multiple threads need to access shared data.

4.2: Learning Outcomes

After completing this unit, you will be able to:

1. Define multithreading and explain its advantages (e.g., responsiveness, resource utilization).

2. Differentiate between a process and a thread.

3. Create threads in Java using the two main methods:

    -   Extending the Thread class.

    -   Implementing the Runnable interface.

4. Describe the life cycle of a thread (New, Runnable, Running, Blocked/Waiting, Terminated).

5. Explain the purpose of synchronization and the synchronized keyword to prevent race conditions.

4.3: Multithreading

Multithreading is a multiprocessor and concurrent programming paradigm that enables multiple threads to run concurrently within a process. Essentially it lets a single program do several things at once, making programs run faster and more responsively, even when they have to do things like I/O or heavy computation work. A thread, the basic unit of CPU utilization, contains a thread ID, a program counter, a set of registers and a stack. Threads created within a process share the code segment, data segment, and operating system resources with all other threads within the same process, hence providing an efficient way of using resources. There are mainly 5 states of thread in Java life cycle namely - New , Runnable, Running, Blocked / Waiting and Terminated. Stage 1: NEW When a thread instance and a thread reference is created using the Thread class or the Runnable interface, it is said to be in a new state. When you call start() method, the thread enters int o Runnable state, which means it is ready to run and chosen by the thread scheduler to start running. When the thread scheduler assigns CPU time to the thread, the thread is moved to the Running state. Threads can enter the Blocked/Waiting state for several reasons, including waiting for I/O operations to complete, needing to acquire a lock, or calling sleep() or wait(). Lastly, a thread goes int o the Terminated state after it has finished executing or when it runs int o an unhandled exception. Java give us many ways to control and manage threads. The code that is executed by the thread is contained in the run() method. The start() method is where the thread actually starts by calling the run() method in the new thread. sleep() : The sleep() method suspends the execution of the thread for the specified amount of time. The join() method is used to wait for a thread to finish executing. yield() — is used to indicate to the thread scheduler that the current thread can relinquish. For shared resource management and avoiding race conditions, synchronization mechanisms are essential (including synchronized blocks and methods). Methods: wait(), notify() and notifyAll() The

basic methods to int er-thread communication between synchronized blocks. Deadlock Problem In Multithreaded Environment: It is a dangerous condition in which two or more threads have blocked indefinitely waiting, each other and needs to be solved. Deadlocks can be avoid ed by correctly managing and synchronizing resources. Thread pools (managed by Executor framework) are an efficient way to manage a set of threads and help avoid the overhead of creating and destroying threads. A Callable is very much like a Runnable, but it can return a value, and it can throw checked exceptions. The Future int erface is for the result of an asynchronous computation, which allows the result to be retrieved once it is available. To create applications that remain responsive and efficient, especially in the world's of networked or server-side processes necessitating concurrent handling. Assembling knowledge of thread management, synchronization, and int er-thread communication is essential for creating resilient and scalable multithreaded applications.

4.4: Network Programming

In simple words, network programming in Java allows you to communicate with other network applications and transfer data between two or more network applications. Yes, Java network programming by ship on the TCP/IP protocol suite. The java. Java provides a rich set of class es for network programming in the java. InetAddress is the class that represents an IP address, which is a numeric label assigned to each device connected to a computer network that uses the Int ernet Protocol for communication. InetAddress class : getLocalHost() and getByName() are some of the methods of the InetAddress class to get the IP address of a host. A Socket class is for the client-side socket, an endpoint for communication between two machines. It defines the IP address and port number of the server which is used to create socket. First, answer why socket class , where Socket class represents a socket for communication between a client and server. The ServerSocket creates a new Socket Object for communication with a client when a client connects to a server. The URL class is used to identify a Uniform Resource Locator, which is a reference to a resource on the web that specifies its location on a computer network as well as a mechanism for retrieving it. Understanding the concept of URLConnection class . It has methods

that can read and write data to the URL. Connection-oriented UDP communication is done with the help of DatagramSocket and DatagramPacket class es. It is a very basic transport layer protocol which provides unreliable, unordered delivery of datagrams. DatagramSocket- Sends and receives datagram packets DatagramPacket- A datagram representing a packet of data Network programming requires things such as setting up sockets, sending and receiving data, handling network exceptions, etc. To read and write data on a network connection, input and output streams are used. The InputStream and OutputStream class es have methods for reading and writing byte streams and the BufferedReader and Print Writer class es have methods for reading and writing char acter streams. Network programming is an important aspect of building distributed applications, web servers, and client-server systems. A device that operates at the lowest level in the OSI model is responsible for packet transmission over these connections. So, these were some of the Pros of using Java.

4.5: Java Database Connectivity (JDBC)

JDBC (Java Database Connectivity) is a Java API that allows Java programs to connect to and int eract with relational databases. JDBC stands for Java Database Connectivity, which is an API for Java programmers to connect with the database. JDBC is divided int o a 2 layered architecture which contains the JDBC API and JDBC drivers. The JDBC API consists of int erfaces and class es that communicate with databases, and JDBC drivers are vendor specific implementations that convert JDBC calls to vendor database commands. The JDBC driver is a piece of software that enables the connection between the Java application and the database. The Types Of JDBC Drivers: Type 1 (JDBC-ODBC Bridge), Type 2 (Native-API Driver), Type 3 (Network Protocol Driver), Type 4 (Thin Driver). Type 1 drivers rely on ODBC to connect to databases; this can be slow and relies on the platform. Type 2 drivers rely on native database libraries, which can be faster but also include platform dependency. Type 3 drivers are a lot easier to work with than type 2 to create because they act as a middleware server with the database, which means they gain portability and scalability. Pure Java Driver (Type 4) — It communicates directly with the database and offers the best performance and platform

independence. In order to open a database connection, you load JDBC driver, generate a connection object, and execute SQL statements. The DriverManager class loads JDBC drivers and returns connection objects. The Connection int erface represents a connection to a database and has methods to create statements, execute queries, and manage transactions. Statement — The Statement int erface is used to execute a static SQL statement and it is suitable for executing a simple SQL statement with no parameters Required, which is a secure way to prevent SQL injection statement, only suitable for executing with no parameters Required of the SQL statement. The ResultSet int erface is an int erface that represents a table of data generated by executing a statement against a database. Especially exceptions related to the database are represented by the SQLException class . The JDBC provides methods for executing SQL statements like SELECT, INSERT, UPDATE, DELETE, etc. To ensure that a series of database operations are executed as a single, atomic unit of work, you can use transactions to group them together. Support for transaction management features, such as commit, rollback, and savepoint s. JDBC: JDBC is very important for developing data-driven applications, as it offers a standard and effective way to connect with relational databases. How JDBC works: JDBC architecture, drivers, and database connectivity in Java.

Multithreading is a fundamental concept in the world of concurrent programming that allows multiple threads to run inside a single process, improving the responsiveness and efficiency of an application. Data from this layer is culturally related to multithreaded Java applications. Thread life cycle, including states such as New , Runnable, Running, Blocked/Waiting, and Terminated through which a thread passes during its lifetime, primarily controls the execution flow of a thread, whereas operations including those in methods such as start(), sleep(), join(), and yield() enable fine-grained control of thread behavior. Synchronization is achieved using synchronized blocks or methods to maint ain data int egrity and avoid race conditions, and int er-thread communication is performed through wait(), notify(), and notifyAll(). A potential pitfall of a multithreading design, deadlock, requires prudent resource management and synchronization techniques to overcome it. The Executor framework is a powerful tool for managing thread pools, optimizing performance by avoid ing the

overhead associated with thread creation and destruction. To enhance Multithreading capabilities, Java provides several int erfaces including the Callable int erface and Future int erface, which allow threads to return values and manage asynchronous computations. In essence, multithreading is crucial for creating responsive, scalable applications, especially in networked or server architectures, where simultaneous execution takes center stage.

Network programming forms the backbone of modern applications, enabling the exchange and int eraction between Java applications and networks. The TCP/IP protocol suite provides a strong foundation for network communication, and Java builds upon that through its features for network programming. The java. The net In the Java programming language, the net package provides a rich set of class es and int erfaces, such as InetAddress, Socket, ServerSocket, URL, URLConnection, DatagramSocket, and DatagramPacket, which enable a network-based application. InetAddress is used to resolve IP addresses in String form, Socket and ServerSocket for establishing client-server communication, URL and URLConnection for fetching a web resource over HTTP, and DatagramSocket and DatagramPacket for making connectionless communication using UDP. Network programming behaviors such as creating sockets, sending and receiving data and handling exceptions during the network operations you will be doing on input and output streams.

Check Your Progress

1. **What are the two ways to create a new thread in Java?**

   …………………………………………………………………
   …………………………………………………………………
   …………………………………………………………………
   …………………………………………………………………..

2. **What problem does the synchronized keyword solve?**

   …………………………………………………………………
   …………………………………………………………………
   …………………………………………………………………
   …………………………………………………………………

4.6: Summary

This unit introduced multithreading, a core Java concept that enables concurrent execution. We learned that a thread is a single path of execution within a larger process, allowing a program to perform multiple tasks simultaneously. We explored the two methods for creating threads: either by extending the Thread class or, more flexibly, by implementing the Runnable interface. We also examined the thread life cycle, understanding that a thread moves between states like New, Runnable, Running, Blocked, and Terminated. Finally, we addressed the critical issue of thread safety. When multiple threads access and modify shared data, we risk "race conditions." The synchronized keyword was introduced as the primary mechanism to lock shared resources, ensuring that only one thread can access them at a time, thus maintaining data integrity.

4.7: Exercises

Multiple Choice Questions

1. Which interface is implemented to create a thread in Java?
   a) Runnable
   b) Serializable
   c) Threadable
   d) Cloneable
   Answer: a) Runnable

2. Which method must be implemented by a class that implements the Runnable interface?
   a) start()
   b) stop()
   c) run()
   d) execute()
   Answer: c) run()

3. What is the state of a thread after it has been created but before start() has been called?
   a) Running
   b) Runnable
   c) New
   d) Terminated
   Answer: c) New

4. Which keyword is used to prevent multiple threads from accessing a shared resource simultaneously?

a) static

b) final

c) synchronized

d) volatile

Answer: c) synchronized

5. Which method is called to begin the execution of a newly created thread?

a) run()

b) start()

c) init()

d) go()

Answer: b) start()

Descriptive Questions:

1. What are the main advantages of using multithreading in an application?

2. Compare and contrast the two methods of creating threads: extending Thread vs. implementing Runnable. Why is implementing Runnable generally preferred?

3. Draw a diagram illustrating the life cycle of a thread and describe what happens in each state (New, Runnable, Running, Blocked/Waiting, Terminated).

4. What is a "race condition"? Provide a simple Java code example that could cause a race condition, and then show how to fix it using the synchronized keyword.

5. Explain the purpose of the join() and sleep() methods in the Thread class.

4.8: References and Suggested Readings

- Balagurusamy, E. (2020). *Programming with Java*. McGraw Hill.

- Schildt, H. (2018). *Java: The Complete Reference*. McGraw Hill.

- Horstmann, C. S. (2021). *Core Java Volume I – Fundamentals*. Pearson.

Block 2: **Java FX Technology**

Unit 5: Introduction to Java FX,

Structure

5.1: Introduction

Welcome to JavaFX! This is a modern software platform for developing and delivering rich client applications that can run across a wide variety of devices. JavaFX is the successor to the older Swing and AWT toolkits, providing a much more powerful and flexible way to build modern graphical user interfaces (GUIs) in Java. It supports a rich set of graphics, media, and animation features, and allows you to style your application with CSS (Cascading Style Sheets). It also introduces FXML, an XML-based language that lets you define your user interface separately from your application logic, making your code cleaner and easier to maintain. This unit introduces the basic building blocks of any JavaFX application: the Stage (the window) and the Scene (the content).

5.2: Learning Outcomes

After completing this unit, you will be able to:

1. Explain the purpose of the javafx.scene.effect package.

2. Apply a simple effect (like DropShadow or GaussianBlur) to a JavaFX node.

3. Understand that effects can be "chained" together to create more complex results.

4. Identify several common effects, such as Reflection, Lighting, Glow, and SepiaTone.

5. Modify the properties of an effect (e.g., changing the radius of a blur or the offset of a shadow).

5.3: Introduction to Java FX

JavaFX represents a significant improvement in building graphical user int erfaces (GUIs) in Java compared to Swing and the Abstract Window Toolkit (AWT). First int roduced by Sun Microsystems (subsequently bought by Oracle), JavaFX was first celebrated as a 2008-era component, giving developers who wanted to develop desktops with rich graphics, embedded media and new programming models a higher-level, more modern way to do it than from what early Java offered with its early emphasis on building complex GUI int erfaces all on its lonesome. Initially, it was an Oracle product, but when it was open-sourced int o OpenJDK 2011, the theoretical changes were made to the code were for every to contribute to the code, allowing for small iterative changes and community development. Initially it was hailed as an answer to Adobe Flash and Microsoft Silverlight, this cross-platform rich-client alternative was capable of the same rich int eractive application creation possibilities, now with the Java ecosystem advantages. When programming languages such as Visual Basic and Visual C++ were int roduced, there was a demand for graphical user int erfaces that were more engaging, allowing int eraction and features that would be visually appealing, that would also run under a large number of operating systems. However, JavaFX, when it landed, came with something of a standalone scripting language (JavaFX Script) which aimed to simplify UI development, thanks to a declarative syntax. But with JavaFX 2.0 (released in 2011), Oracle returned to a pure-Java-API approach, ditching the separate scripting language in favor of regular Java code feathered with builder patterns and fluent APIs. Thus JavaFX became a tangible platform for the existing pool of Java developers based on the extensive familiarity with Java and some of the ability to do modern UI development. The long and short of it is simply this: JavaFX was never really about the technology — it was a case study for Oracle that Java was still relevant, even in the midst of an explosion of web and mobile technologies. : JavaFX was the first stone in that rich client mountain: it established the right architecture

for taking Java out of the server and int o both the desktop stacks. The major functionality and improvements were rolled out with every new release over a series of reworks. The core Java API was developed in JavaFX 2.0; JavaFX 8 (along with Java 8) int egrated more with the Java Development Kit (JDK), and the more recent releases enhanced performance, added new UI controls, extended platform support.So this makes 2018: a new major shift for JavaFX: with Java 11, it was decoupled from the JDK. While this added some extra steps when you wanted to include it in your projects, this modularization allowed JavaFX to release its libraries independently of the overall Java platform release cadence. But what do you know, all this was possible by JavaFX for Java which was released on December 3, 2008, and eventually led to what we have today, a complete mature framework for building rich cross platform applications with powerful graphics, multimedia support and advanced UI components. And its evolution is a window int o some of the most significant trends in software development overall, including the shift toward more declarative programming models, the rising need for rich user experiences, and the new need for cross-platform compatibility as an ever more heterogeneous computing landscape emerges. Having a clearer understanding of what led us to here, we now have the context better to look at its present capabilities and its role in the wider Java ecosystem before looking at its feature set, architecture and use in modern application development.

5.4: JavaFX Application Structure

JavaFX application is divided hierarchically int o three main components known as Stage, Scene and nodes. We need to import **javafx.application.Application** class in every JavaFX application. This provides the following life cycle methods for JavaFX application.

- o public void init()
- o public abstract void start(Stage primaryStage)
- o public void stop()

in order to create a basic JavaFX application, we need to:

1. Import **javafx.application.Application** int o our code.
2. Inherit **Application** int o our class .
3. Override **start()** method of Application class .

Stage

Stage in a JavaFX application is similar to the **Frame** in a Swing Application. It acts like a container for all the JavaFX objects. Primary Stage is created int ernally by the platform. Other stages can further be created by the application. The object of primary stage is passed to **start** method. We need to call **show** method on the **primary stage object** in order to show our primary stage. Initially, the primary Stage looks like following.
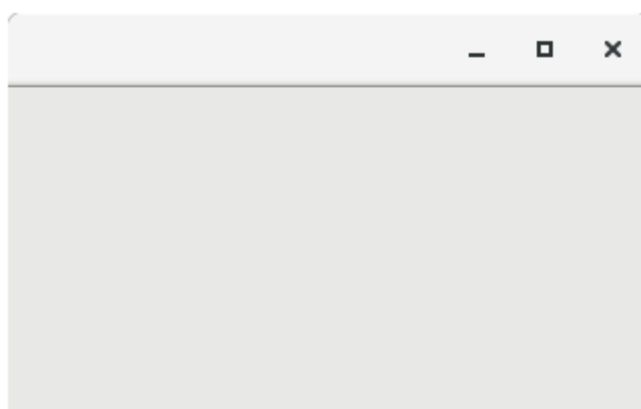
Figure 5.1 Stage

However, we can add various objects to this primary stage. The objects can only be added in a hierarchical way i.e. first, scene graph will be added to this primaryStage and then that scene graph may contain the nodes. A node may be any object of the user's int erface like text area, buttons, shapes, media, etc.

Scene

Scene actually holds all the physical contents (nodes) of a JavaFX application. **Javafx.scene.Scene** class provides all the methods to deal with a scene object. Creating scene is necessary in order to visualize the contents on the stage.

At one instance, the scene object can only be added to one stage. In order to implement Scene in our JavaFX application, we must import **javafx.scene** package in our code. The Scene can be created by creating the **Scene** class object and passing the layout object int o the Scene class constructor. We will discuss Scene class and its method later in detail.

Scene Graph

Scene Graph exists at the lowest level of the hierarchy. It can be seen as the collection of various nodes. A node is the element which is visualized on the stage. It can be any button, text box, layout, image, radio button, check box, etc.

The nodes are implemented in a tree kind of structure. There is always one root in the scene graph. This will act as a parent node for all the other nodes present in the scene graph. However, this node may be any of the layouts available in the JavaFX system.

The leaf nodes exist at the lowest level in the tree hierarchy. Each of the node present in the scene graphs represents class es of **javafx.scene** package therefore we need to import the package int o our application in order to create a full featured javafx application.
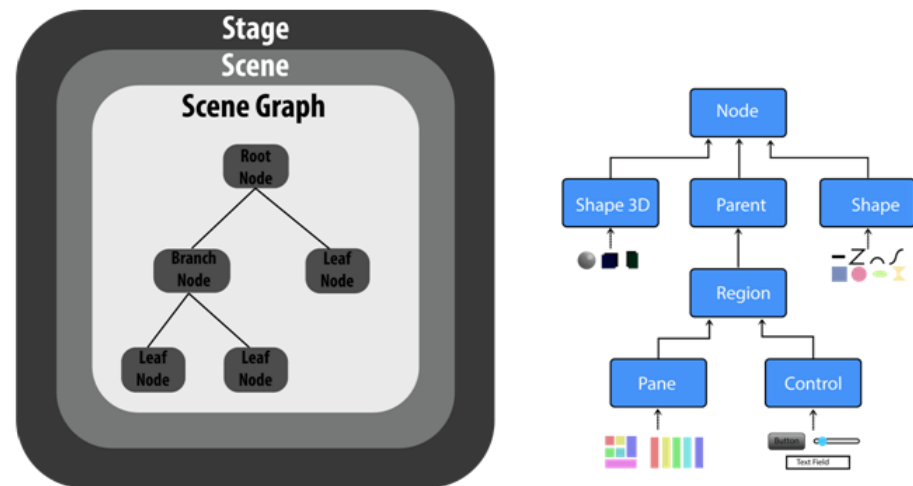
Figure 5.2 Scene Graph

Historical Context:

Java's history of developing graphical user int erfaces has undergone an evolution driven by paradigms shifts in technology and development as well as developer and user expectations. This started with the Abstract Window Toolkit (AWT), the original GUI toolkit that Java shipped with the first version of Java in 1995. AWT offered a basic set of UI components that mapped directly to native platform components, in what is known as a "heavyweight" approach. Although this method allowed applications to preserve the appearance and behavior of the underlying operating system, it limited the level of customization and appearance consistency across different platforms. Moreover, the component set of AWT was quite limited with basic components only buttons, text fields and basic containers etc. These factors led to the

creation of Swing, which was released in 1997 as part of the Java Foundation Class es (JFC). Previously, Swing was a major improvement because it adopted a "lightweight" architecture, which meant that in most cases each of Swing's components were drawn using Java's own rendering engine instead of native components. That was way more flexible, had much richer component set, and much more consistent behavior cross platform. From this, Swing adopted the pluggable look-and-feel system to enable applications to look the same regardless of the underlying operating system or adopt the native look and feel when needed. Swing remained the de facto GUI toolkit for over a decade with commercial and enterprise applications building on thousands of Swing-based applications and establishing the baseline for user int erface design in the Java ecosystem. But as web and mobile applications grew and as users experienced more advanced user int erfaces, expectations were updated for desktop applications as well. For modern users, rich animations, seamless multimedia int egration, hardware-accelerated graphics—and more visually engaging experiences—were all things that pressed Swing beyond its initial design parameters. These evolving expectations, together with improvements in graphics hardware and new rendering technologies, set the stage for the arrival of JavaFX. JavaFX was first developing as "Project F3" (Form Follow Function) within Sun Microsystems, were first announced as a public product in 2007 and first released in 2008. First iteration (JavaFX 1. x), which had a dedicated scripting language (JavaFX Script), that allowed you to describe user int erfaces in a declarative manner. It was a radical departure from Swing's imperative programming model. Another focus was the int egration of rich media and the added support for animation, which positioned JavaFX as competition for Adobe Flash and Microsoft Silverlight in the arena of Rich Int ernet Applications (RIA). Oracle bought Sun Microsystems in 2010, and for a while there it didn't look good for JavaFX. But then Oracle established its real commitment to the platform when it announced a massively ambitious roadmap. JavaFX 2.0, int roduced in 2011, was a pivotal change, dropped the separate scripting language and used a standard Java API. This move brought JavaFX int o closer alignment with mainstream Java development practices, but without sacrificing the advanced graphics and animation features available in the platform. This

evolution continued with JavaFX 8 in 2014, which aligned versioning with the Java SE platform and provided complete int egration for JavaFX; included as part of the JDK. [More changes that includes UI controls, 3D, touch] This release added a number of new UI controls, better 3D graphics support, and improved touch capabilities: an acknowledgment of the rising significance of touch-enabled devices.

For example, in 2018, the biggest milestone was that JavaFX got decoupled from the JDK with Java 11 and became an independent module under the OpenJFX project. Doing so gave JavaFX the freedom to grow on its own timetable, separate from the release schedule of the core Java platform. As each GUI framework evolved, they improved upon their predecessors' limitations and adapted to the changing technological landscape and user expectations. AWT offered some primitive platform native components, Swing better flexibility and more components, and JavaFX hardware acceleration, modern skinning via CSS, richer animation, and full multimedia support. It also signals an evolution in mindset, moving from imperative programming and dense code in AWT and Swing, to the emphasized declarative design encouraged by JavaFX, especially with FXML for UI definition.

5.5: Positioning in the Modern UI Landscape

JavaFX maint ains a unique position in the wealth of user int erface technologies available to developers today; indeed it reflects its technical prowess with a strategic value proposition. JavaFX and its place among the alternatives for building GUIs (including web development, native platform toolkits and the other cross-platform options) which gives you insight int o this position. This is one of the many strong point s of JavaFX, the ability to be able to create true native applications with the same behaviour across operating systems. While most web applications rely on a browser runtime, JavaFX applications can include all the required runtime components and be distributed as standalone executables. This is still useful in cases where tight int egration with the OS, offline capabilities, or access to local system resources is needed. JavaFX also boasts a cross-platform architecture that enables it to run not only on Windows, but also on macOS and Linux, and even to some extent, mobile platforms, which can be a big plus when building applications that need to run in

heterogeneous computing environments. For organizations that have a variety of technology ecosystems, or for those that are creating software for distribution to people who may be using any number of operating systems, they can rely on one code base rather than maint ain distinct implementations for each platform. This cross-platform capability puts JavaFX in competition with frameworks such as Qt, Electron, and Flutter — each of which has its own take on the dilemma of cross-platform development. JavaFX can be seen as a natural enterprise extension to companies that have invested heavily in Java technology. Java is pervasive in the enterprise, with many organizations having established Java development skills, build pipelines, security practices, and deployment workflows. JavaFX taps int o this already well-established ecosystem, providing these organizations with the ability to develop complex, sophisticated desktop applications without a new programming language, or a completely different programming approach. This int eroperability with the wider Java ecosystem, including compatibility with tools, frameworks, build tools, and IDEs, offers a unified programming experience that sets JavaFX apart from other solutions that may require the adoption of entirely new technology stacks. Today the User Int erface of web applications are heavily inspired by web technologies and frameworks like React, Angular, Vue. js includes much of modern user int erface development. JavaFXacknowledges this fact by providing the capability to embed web content int oapplications with the WebView component, which is similar to embedding a web browserinside an application. This hybrid approach allows the developers to leverage the best features of web technologies for content rendering while merging it with the platform int egration and performance advantages provided by a native application framework. Additionally, JavaFX adopts concepts from modern web development, as seen in the use of CSS for styling and FXML for separating presentation and logic. These features also make it easier for developers who are familiar with these types of technologies to work with the stack, and align with the broader industry trend toward defining UIs in a declarative fashion and separation of concerns. JavaFX shines above other technologies when it comes to data-driven enterprise applications. You are still an Editor for importing concepts, concepts int o which the framework can be used to bind the connection of really, making the int erface responsive, in

which case you can update the data when some data is actually changed. When these two powerful technologies are combined together, it creates the perfect platform for business applications requiring data visualization, analysis, and manipulation due to Java's rock-solid data processing capabilities and a wealth of connectivity options to databases and services. The Introduction and success of Electron, which bundles web applications with a Chromium runtime to create desktop applications, has reshaped the desktop application landscape. Electron has revolutionized the world of desktop apps but comes with few drawbacks such as performance and resource hogging but JavaFX is one good alternative. JavaFX applications tend to be smaller in terms of size and resource usage, compared to Electron applications that require an entire web browser engine to be included. This efficiency is crucial for applications that run on systems with limited resources or efficiency-critical applications. JavaFX stands out with its excellent multimedia and graphics support as well. Positioning it well, for applications which needs rich visual experiences, is its scene graph architecture, hardware-accelerated rendering pipeline and built-in support for animation, 3D graphics, and a variety of media formats. The rich media support and scene-graph architecture allow JavaFX to be used for data visualization, demonstrations, educational software as well as creative software such as keyframing tools where primitives must render dynamically. As web applications have grown more complex, the lines between web and desktop applications have become less distinct. JavaFX recognizes this convergence with CSS styling, the FXML markup language for UI definition, and WebView for web content int egration. It still has the power of a compiled language and a native runtime, providing performance and security char acteristics that manage to be hard to come by through an entirely web-based solution.

5.6: Core Philosophy and Design Principles

JavaFX was designed based on a set of core philosophy and design principles which continue to influence its design and usage. These principles are drawn from the lessons of past Java UI frameworks as well as future directions in application development in a more heterogeneous and dynamic computing ecosystem. One of the principles that the design philosophy around JavaFX is built on is the need for expressive and declarative user int erface construction. In contrast with the more imperative programming model of AWT and Swing, where int erfaces were created almost exclusively by procedural code, this is a major advancement. JavaFX In a way, also embraces a more declarative paradigm, especially with FXML for defining user int erfaces in an XML-based markup language.
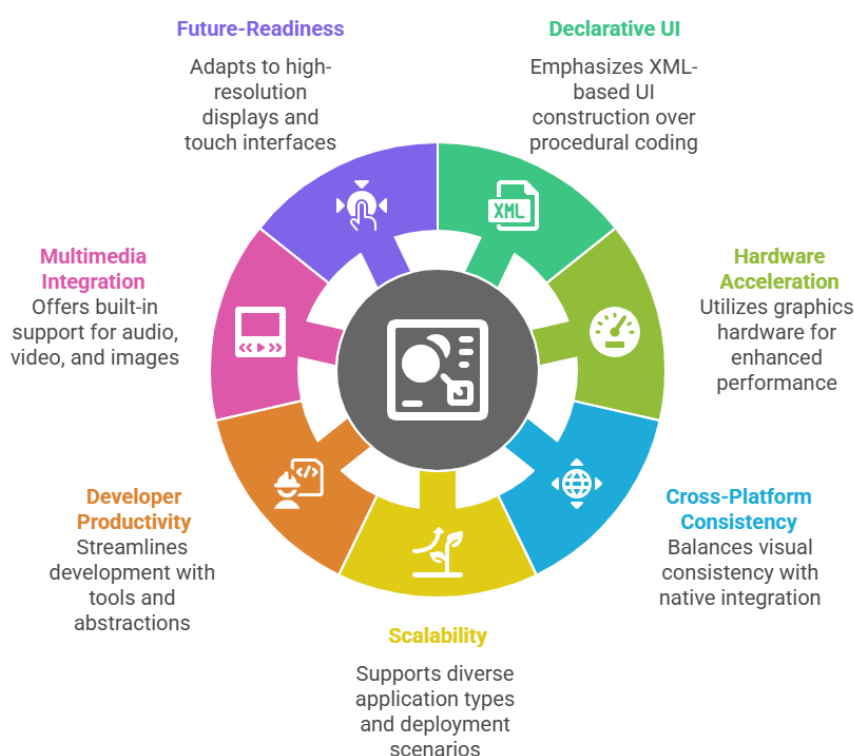
Figure 5.3 Java FX Design Principles

While draft.is or TiddlyWiki is structured as an application — an int erface containing all its own logic — React.js separates UI structure from application logic. As we will touch on the declarative approach further above the visual and aesthetic, the declaration based approach even flows over the structuring to the aesthetic, so JavaFX uses CSS to allow usto visually adjust UI elements. This choice enhances the broad

knowledge base surrounding CSS, both for web developers and web designers, while simultaneously providing a robust and standard way to build visually striking apps without needing to change int ernal code. The ability to apply multiple stylesheets and to work with dynamic styles also help towards building int erfaces that are visually coherent and adaptive. Another one of the core concepts is hardware acceleration out of the box — JavaFX has been designed from the very beginning to maximize the potential of existing graphics hardware by way of its Prism rendering pipeline. Using this method, artists can construct rich animations and render dense scenes, with pixel-perfect accuracy regardless of size (including very large displays). By providing a graphics pipeline that abstracts the int eraction with hardware, JavaFX enables developers to write visually and functionally rich applications without the need of detailed knowledge with any specific graphics system while utilizing the hardware when it is available. JavaFX is also designed for cross-platform consistency while still respecting platform conventions. Unlike previous approaches that tended to leave developers choosing one or the other between cross-platform visual consistency and native int egration, JavaFX attempts to balance these tradeoffs. The platform differs between what is showable patterns and functional behaviors when appropriate, but provides a uniform model and idea of contributions to underlying systems. This may sound like a no-brainer, but it applies to accessibility too: JavaFX is built to work with any type of assistive technology across multiple platforms, so applications can be used by people of all abilities. JavaFX represents the idea of scale in terms of the various kinds of application and deployment cases. Its architecture handles everything from simple forms-based business applications, to data visualization tools, to complex maps with rich graphics. It can be used for standalone desktop applications, for web deployment through Java Web Start (in previous versions), or for embedded systems applications. This is implemented through modules, meaning that you can only add the necessary components for the given needs of the application. Pretty much any Java SE application can contain JavaFX components, and JavaFX itself is available as an importable Java library. This allows developers to take advantage of their existing investment in Java technology as they learn and adopt the modern UI features of JavaFX. The framework offers initial support to int egrate

Swing components when needed, allowing upgrading of older applications to be done in a gradual fashion. Another major aspect of Flutter is its WebView component, which allows for int egration of browser content, acknowledging the significance of web technologies in current applications. Developer productivity has, in fact, hugely impacted JavaFX's design. Then I also mention properties binding (or whatever name it's got inside your own UI library, with property IDs that can be only bound in a declarative way from the data model while automatically redoing the view upon data changes so that it is not needed to do the same manually in code), which cut the amount of boilerplate code and up is not prone to consistency errors, as well as getting rid of a lot of boilerplate code. In the same way, the animation framework does not expect you to do complex mathematical calculations, instead, it offers high-level abstractions for creating rich transitions and effects. The event handling system, which follows consistent patterns across various component types, also increases developer efficiency as the learning curve is lowered. In addition, JavaFX follows the design/developer collaboration approach by supporting tools like Scene Builder, which is a visual design environment that produces FXML that can be used directly in applications. This strategy acknowledges the reality of modern application development, where implementation and design specialists increasingly collaborate. This separation of concerns in FXML and CSS makes it easy for the designer to work on all of the visual aspects without needing to focus on how this will all fit in the application logic. The next core design principle is multimedia int egration, and this was a crucial consideration in the development of JavaFX, which offers first-class support for audio, video, and images without the need for additional third-party libraries. JavaFX also has built-in support for images, audio, and video, which reduces the need for external libraries or plugins for common media operations to develop rich client applications. It even extends to 3D content, because JavaFX natively supports 3D objects and scenes as part of its out-of-the-box arsenal. JavaFX finally reflects the idea of future-readiness with a number regarding display technologies and help for touch int erfaces and new int eraction patterns. The platform was created with an eye towards trends for high resolution displays, touch capable devices, and animated user experiences. This future-proofing helps guarantee that JavaFX-

built applications do not go out of date as computing environments persist in metamorphosing.

5.7: Core Features and Capabilities of JavaFX
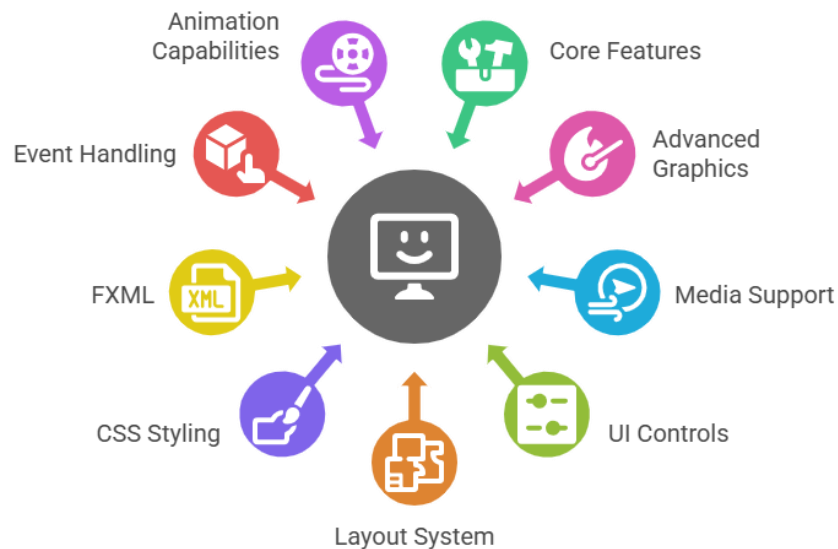
Figure 5.4 Core Feature and Capability

These capabilities are just a glimpse int o the powerful tools JavaFX offers for developing high-performance, cross-platform applications with stunning graphics and UI. The core of it is a scene graph architecture, where graphical elements are arranged in a hierarchical way that allows for quick rendering and int eraction response. It is the foundation on which JavaFX builds its approach of arranging UI components, layouts, and custom visual objects as a hierarchy of nodes in a scene graph. It includes a comprehensive library of pre-built UI controls including buttons, text fields, tables, trees, char ts, and more. These controls match modern UI patterns and expectations like animation, visual effects, and styling (including CSS). The styling system approach that JavaFX int roduced is a huge improvement compared with other UI frameworks in Java, enabling developers to decouple the visual aspect from the application logic and to deliver visually unique applications without touching their int ernal code. Prism is the platform's rendering engine that uses hardware acceleration to maint ain graphics performance, especially for animations and effects. This hardware-accelerated pipeline allows JavaFX apps to provide visually stunning experiences even for complex scene

rendering and high resolution content. Along with these visual features, JavaFX has full multimedia support with built-in class es for images, audio, and video. Such int egrated support means no additional libraries and APIs are needed for working with common media formats, enabling the rapid development of complex content-rich applications. JavaFX, on the other hand, offers built-in visual support for 3D scenes, enabling developers to construct and manipulate three-dimensional objects as needed, as well as leveraging the same APIs for traditional 2D int erfaces. JavaFX's binding framework, which allows UI elements to be declaratively bound to underlying data models, is another differentiator. When data changes, the UI automatically updates and you get to write a little less boilerplate to synchronize presentation code with data to avoid inconsistencies between the two. This two-way binding goes from property to property across the framework and allows creating more adaptive, data-driven applications. JavaFX also provides a declarative approach to UI definition via FXML, an XML-based markup language. The accompanying Scene Builder tool offers a similar, visual design
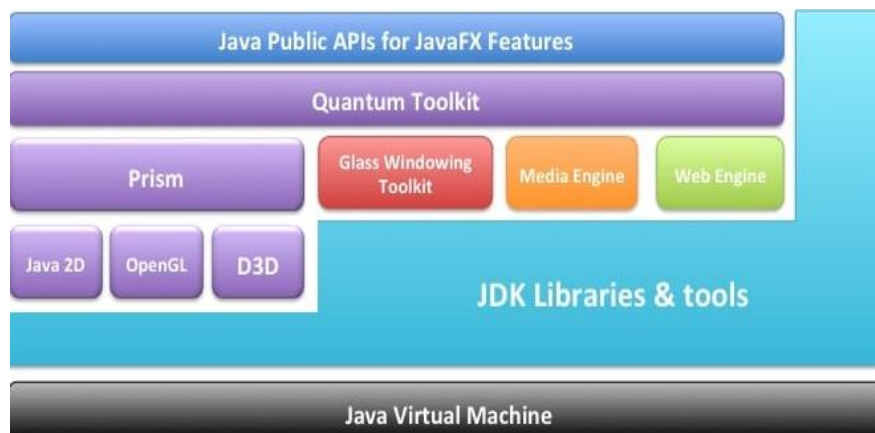
Figure 5.5: JavaFX Architecture
Source: https://static.packt-cdn.com/

experience for building out JavaFX int erfaces, outputting FXML declarations that can be used out of the box in applications. JavaFX also has int egrated WebView, which embeds a web browser engine, int o the content. This allows applications to render HTML, run JavaScript, and communicate with web applications, essentially merging desktop and web technologies. JavaFX Int egration and PerformanceJavaFX Core FeaturesThe combination of all of these core features makes JavaFX a powerful tool for building modern applications that have the

performance and int egration char acteristics of traditional native applications complemented with the more **advanced visual richness and application int eraction models that users** are becoming accustomed to.

5.8: Scene Graph and UI Components
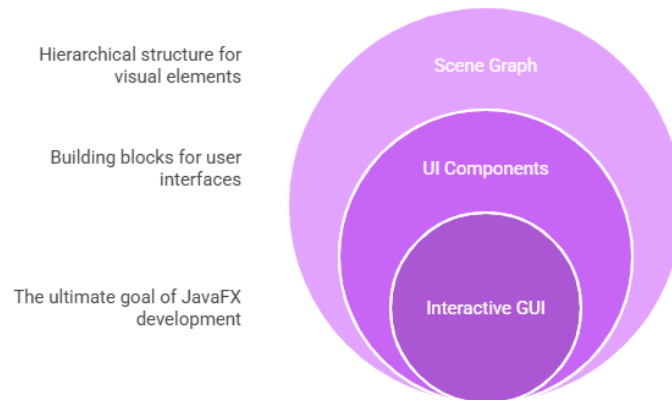
Figure 5.6 UI Components

At the center of JavaFX's rendering architecture is a scene graph, which is a hierarchical structure that represents all of the visual elements in a single application. This approach to constructing user int erfaces is a monumental shift from the Java UI frameworks that preceded it and supports many advanced features of JavaFX. A scene graph is organized as a tree where each node in the tree is either a visual element, a group of visual elements, or some operation (a transformation or an effect) applied to its children. Such a hierarchical organization lends itself well to the compositional nature of user int erface as-built (composite components are built of more simple components). In JavaFX, the scene graph starts with a Stage that serves as the top-level container, usually a window in desktop applications.Scenes graph structureA Stage has exactly one Scene, which holds the root node of the scene graph. From this root, you have a tree of nodes extending (or a graph if you want to be technical) for all visual elements in the int erface. Node class Diagram The Node class is the root of all objects in the scene graph and contains common properties and behaviours for positioning, transformation, effects, event handling, and user int eraction. JavaFX divides its nodes int o some categories: shapes (Rectangle, Circle, Path), controls (int eractive components like Button, TextField and TableView), containers (layout components like HBox,

VBox, and BorderPane), media nodes (ImageView, MediaView) and web content (WebView). Note that Group nodes are also used to combine multiple nodes int o a single node which can be executed as an atomic unit. These various node types act as building blocks for crafting int erfaces that can range from basic forms to elaborate visualizations. The scene graph architecture provides many strong benefits to UI development. First, it is a natural model for building complex int erfaces using simple components.
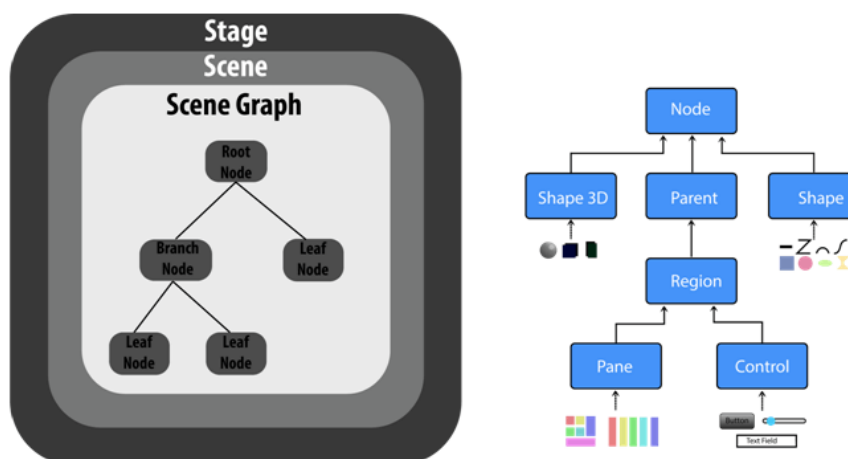
Figure 5.7 Stage, Scene & Graph

Users can compose new types of components from merely existing nodes, transformations and effects, and custom behaviors. Second, the hierarchy aids efficient rendering with culling (trees that are far away from view aren't rendered) and dirty region (only redrawing the sections that have changed.) The JavaFX runtime will automatically take care of these optimizations, so developers can focus on writing their complex int erfaces without having to have knowledge of the rendering optimizations. Third, the scene graph provides a single model for transformations and transition, which simplifies animation and visual effects. Any node in the graph can have properties such as position, rotation, scale, and opacity animated to produce complex visual behaviors with minimal code. The Scene graph is the core hierarchical structure upon which JavaFX UI components are built, providing a rich toolkit for building applications. Components can be simple elements or complex, data-driven controls. JavaFX provides primitive shapes (like Rectangle, Circle, Line, Path, etc.) in its most simple form for building custom graphics. Text nodes can display formatted text with a variety of fonts, styles, and effects. The framework offers a wide range of layout components (HBox, VBox,

BorderPane, GridPane, FlowPane, etc.) that position its children based on different spatial configurations and are responsive to size changes. JavaFX provides a rich set of controls that implement common UI patterns for user int eraction. These include basic controls like Button, Label, TextField, PasswordField and CheckBox. Selection controls include ChoiceBox, ComboBox, ListView, TreeView, and TableView. That said, JavaFX has Slider, ProgressBar and ScrollBar for numerical input. Date selection is managed by DatePicker and complex text entry is provided by TextArea and HTMLEditor. Higher-level components include the list of char t types (PieChar t, LineChar t, BarChar t, etc.) for data visualization, TreeTableView for hierarchical data representation, and Pagination to split data int o pages. Basic int eraction patterns such as alerts, confirmation requests, and custom modal int erfaces are provided by the dialog components. They follow common patterns for styling, int eraction, and customization. The component exposes its properties, which can be bound to application data, configured programmatically, or set with FXML. Components emit events when users int eract with them as part of a unified event model that greatly simplifies the implementation of int eractive behaviors. JavaFX controls are designed to be functional and leave it up to the programmer to decide how it should look. Each control provides a complete implementation of its int ended functionality out-of-the-box, with developers able to extensively customize appearance and behavior. This customization can take place at several levels: CSS styling, properties set in code, changing the control's cell factory (for list-based controls), or by building completely new controls with subclass ing or composition. This versatility enables developers to design functional yet visually improved int erfaces. JavaFX's implementation of UI components is designed to be accessible, allowing its applications to be compatible with screen reader software and other assistive technologies. Find out how JavaFX implements appropriate roles and provides accessibility information, contributing to the ability for applications built with JavaFX to be usable by people of varying abilities. Context: scene graph and component model ⇒ declarative UI construction Unlike earlier frameworks, where developers imperatively controlled low-level graphics contexts, JavaFX developers specify the desired contents and structure of the int erface. The framework abstracts away the specifics

of rendering, layout, and event propagation, resulting in cleaner, more maint ainable, and less error-prone code.

Check Your Progress

1. **In a JavaFX application, what is the main container for all content, analogous to a window?**

    ……………………………………………………………
    ……………………………………………………………
    ……………………………………………………………
    ……………………………………………………………

2. **What is FXML, and why is it useful?**

    ……………………………………………………………
    ……………………………………………………………
    ……………………………………………………………
    ……………………………………………………………

5.9: Summary

This unit introduced JavaFX as the modern Java platform for creating rich graphical user interfaces. We learned that every JavaFX application is built upon a central metaphor: a theater. The Stage is the main window (the theater itself), and it holds a Scene, which contains all the visible content (the play). We identified that the main class of an application must extend javafx.application.Application and override the start() method, which is the main entry point. Finally, we introduced FXML, an XML-based markup language that provides a powerful way to separate the UI design (the "what") from the application's behavior (the "how"), leading to more modular and maintainable code.

5.10: Exercises

Multiple Choice Questions

1. What is the top-level container or window in a JavaFX application?
    a) Scene
    b) Stage
    c) Pane
    d) Application
    Answer: b) Stage

2. Which class must every JavaFX main application class extend?
a) javax.swing.JFrame
b) javafx.application.Application
c) javafx.ui.Window
d) java.applet.Applet
Answer: b) javafx.application.Application

3. What is the main entry point method for a JavaFX application, which you must override?
a) main()
b) init()
c) start()
d) run()
Answer: c) start()

4. The content inside the Stage (the window) is held by the:
a) Scene
b) Root
c) Container
d) Layout
Answer: a) Scene

5. What file-based XML language is commonly used to define JavaFX user interfaces declaratively?
a) HTML
b) CSS
c) FXML
d) SwingXML
Answer: c) FXML

Descriptive Questions:

1. Explain the relationship between the Stage, Scene, and "Scene Graph" in JavaFX. Use a clear analogy (like a theater).

2. What is FXML? Discuss at least two major advantages of using FXML for UI design instead of writing pure Java code.

3. Describe the basic life cycle of a JavaFX application. What is the purpose of the init(), start(), and stop() methods?

4. How does JavaFX differ from its predecessor, Swing? List and explain three key differences.

5. Write the complete code for a minimal "Hello World" JavaFX application that shows a single window with that text.

5.11: References and Suggested Readings

- Dea, J., Weaver, J., & Sharan, S. (2014). *JavaFX 8: Introduction by Example*. Apress.

- Schildt, H. (2018). *Java: The Complete Reference*. McGraw Hill.

Block 2: **Java FX Technology**

Unit 6: Java 2D Shapes, Colors and Text

Structure

6.1 Introduction

6.2 Learning Outcomes

6.3 Java 2D Shapes, Colors, and Text

6.4 Customizing Shapes with Colors and Strokes

6.5 Combining Shapes, Colors, and Text for Int eractive Graphics

6.6 JavaFX 2D Shapes

6.7 JavaFX Line

6.8 JavaFX Circle

6.9 JavaFX Graphical Effects and Transformations

6.10 Summary

6.11 Exercises

6.12 References and Suggested Readings

6.1: Introduction

Now that we understand the basic structure of a JavaFX application (Stage and Scene), we can start adding content. The javafx.scene package is the foundation for the **Scene Graph**, which is the tree of all visual elements in your scene. This unit focuses on the fundamental components of 2D graphics. We will explore how to create and manipulate basic 2D shapes like lines, rectangles, circles, and polygons. We'll also learn how to add visual style by applying colors, gradients, and strokes (outlines) using the Paint class. Finally, we will cover how to render and customize text, controlling its font, size, and position.

6.2: Learning Outcomes

After completing this unit, you will be able to:

1. Understand the concept of the JavaFX Scene Graph.
2. Draw basic 2D shapes such as Line, Rectangle, Circle, and Ellipse.
3. Apply colors and gradients to shapes using the Color and Paint classes.
4. Customize the appearance of shapes by setting stroke (outline) properties.
5. Create and style text using the Text class.

6. Position elements within a Pane or Group using X/Y coordinates.

6.3: Java 2D Shapes, Colors, and Text

Java offers a strong 2D graphics API within the java. awt and javax. swing packages that help the developer customize/add shapes, colors, and text in their graphical applications. These and other functionality to draw basic shapes like lines, rectangles, ovals, polygons can be achieved using class es called Graphics and Graphics2D. The Graphics2D class is an extension of Graphics class , which contains more sophisticated control over geometry, coordinate transformations, color management, and text layout. For instance, by overriding the paint Component method and using Graphics2D on a Swing component, you can draw a rectangle and an ellipse with varying colors and stroke widths.

```java
import javax.swing.*;
import java.awt.*;

public class ShapeDrawing extends JPanel {
    @Override
    protected void    paint Component(Graphics g) {
        super.paint Component(g);
        Graphics2D g2d = (Graphics2D) g;

        // Set color and draw a rectangle
        g2d.setColor(Color.BLUE);
        g2d.fillRect(50, 50, 100, 70);

        // Set stroke and draw an oval
        g2d.setColor(Color.RED);
        g2d.setStroke(new      BasicStroke(3));
        g2d.drawOval(200, 50, 100, 70);
    }

    public static void  main(String  [] args) {
        JFrame frame = new      JFrame("Java 2D Shapes");
        frame.add(new      ShapeDrawing());
        frame.setSize(400, 200);
```

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
    }
}
```

6.4: Customizing Shapes with Colors and Strokes

You can customize the shapes with colours, gradients, and stroke styles and paint it as per your need. The Color class contains some predefined colors and it can also create custom colors based on RGB. Additional styles, such as smooth color transitions and different line weights can be done with the class es GradientPaint and BasicStroke. In the following example, we apply a gradient fill to a rectangle, and use a dashed stroke for a line.

```
g2d.setPaint (new GradientPaint (50, 50, Color.BLUE, 150, 120,
Color.CYAN, true));
g2d.fillRect(50, 50, 100, 70);

float [] dashPattern = {10, 5, 2, 5};
g2d.setStroke(new     BasicStroke(3, BasicStroke.CAP_ROUND,
BasicStroke.JOIN_BEVEL, 1, dashPattern, 0));
g2d.setColor(Color.BLACK);
g2d.drawLine(50, 150, 200, 150);
```
This snippet demonstrates how Java 2D enables **smoother, more visually appealing** drawings beyond basic shapes.

6.5: Combining Shapes, Colors, and Text for Int eractive Graphics

This implies that int egrating these elements also means that the developers can reap visual applications of anything from drawing apps through games and visualizations. As an example, we can dive to a real-world use case of a dashboard visualization, which has bars inside bars filled with gradients, custom strokes outline and anti aliased text labels. It is powered for them to create visually enhanced and int eractive UI components.

6.6: JavaFX 2D Shapes

JavaFX provides a rich set of **Shape** class es in the package javafx.scene.shape.

These shapes are **nodes** that can be added directly to a Scene or inside layout   containers.

You can style them with properties like fill, stroke, strokeWidth, etc.

Common 2D shapes in JavaFX include:

Shape Class	Description
Line	A straight line between two point s (startX,startY) and (endX,endY).
Rectangle	A rectangle with specified width, height, and optional arc for rounded corners.
Circle	A circle defined by center coordinates (centerX,centerY) and a radius.
Ellipse	An ellipse defined by center coordinates and two radii (radiusX, radiusY).
Polygon	A shape with multiple sides defined by point s.
Polyline	Similar to Polygon but not closed.
Arc	A section of an oval or circle defined by a start angle and length.

JavaFX provides the flexibility to create our own 2D shapes on the screen .There are various class es which can be used to implement 2D shapes in our application. All these class es resides in javafx.scene.shape package.

This package contains the class es which represents different types of 2D shapes. There are several methods in the class es which deals with the coordinates regarding 2D shape creation.

What are 2D shapes?

In general, a two dimensional shape can be defined as the geometrical figure that can be drawn on the coordinate system consist of X and Y planes. However, this is different from 3D shapes in the sense that each point of the 2D shape always consists of two coordinates (X,Y).

Using JavaFX, we can create 2D shapes such as Line, Rectangle, Circle, Ellipse, Polygon, Cubic Curve, quad curve, Arc, etc. The class javafx.scene.shape.Shape is the base class for all the shape class es.

How to create 2D shapes?

As we have mentioned earlier that every shape is represented by a specific class of the package javafx.scene.shape. For creating a two dimensional shape, the following instructions need to be followed.

1. Instantiate the respective class : for example, Rectangle rect = new Rectangle()

2. Set the required properties for the class using instance setter methods: for example,

```
rect.setX(10);
   rect.setY(20);
   rect.setWidth(100);
   rect.setHeight(100);
```
3. Add class object to the Group layout: for example,

```
Group root = new      Group();
   root.getChildren().add(rect);
```
The following table consists of the JavaFX shape class es along with their descriptions.

Shape	Summary
Line	Connects two point s (X,Y) on a 2D plane. Use javafx.scene.shape.Line.
Rectangle	Four-sided figure with equal opposite sides and right angles. Use javafx.scene.shape.Rectangle.
Ellipse	A curve with two focal point s; distance sum to focal point s is constant. Use javafx.scene.shape.Ellipse.
Arc	A segment of a circle or ellipse. Use javafx.scene.shape.Arc.
Circle	A special ellipse with coinciding focal point s. Use javafx.scene.shape.Circle.

Shape	Summary
Polygon	A closed shape formed by joining multiple line segments. Use javafx.scene.shape.Polygon.
Cubic Curve	A 3rd-degree curve in the XY plane. Use javafx.scene.shape.CubicCurve.
Quad Curve	A 2nd-degree curve in the XY plane. Use javafx.scene.shape.QuadCurve.

Note: JavaFX provides dedicated class es in javafx.scene.shape for creating and displaying a wide range of 2D geometric shapes, each with properties you can customize and render on a scene.

JavaFX Shape Properties
All the JavaFX 2D shape class es acquires the common properties defined by **JavaFX.scene.shape.Shape** class . In the following table, we have described the common shape properties.

Property	What it Does	Setter Method
fill	Fills the int erior of the shape with a paint or color.	setFill(Paint)
smooth	If true, smooths the edges of the shape.	setSmooth(boolean)
strokeDashOffset	Defines dash pattern offset for creating dashed outlines.	setStrokeDashOffset(double)
strokeLineCap	Sets the style of the line's end caps (e.g., butt, round, square).	setStrokeLineCap(StrokeLineCap)

Property	What it Does	Setter Method
strokeLineJoin	Sets the style of the joint where two lines meet.	setStrokeLineJoin(StrokeLineJoin)
strokeMiterLimit	Limits the distance between inner and outer corner point s of a joint .	setStrokeMiterLimit(double)
stroke	Sets the color or paint used for the outline (border) of the shape.	setStroke(Paint)
strokeType	Determines where the stroke is drawn (inside, outside, or centered).	setStrokeType(StrokeType)
strokeWidth	Sets the thickness of the outline.	setStrokeWidth(double)

These properties let you style shapes in JavaFX by customizing their fill color, outline color and width, smoothing, dash patterns, and how line ends and joins are rendered.

setFill Example

setStroke Example

setStrokeWidth Example
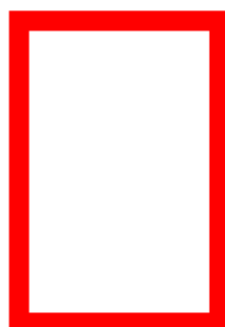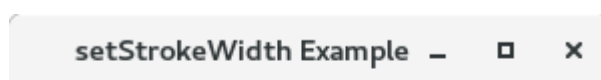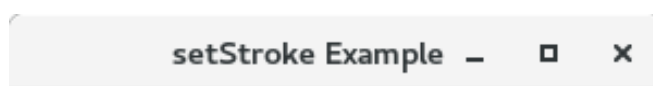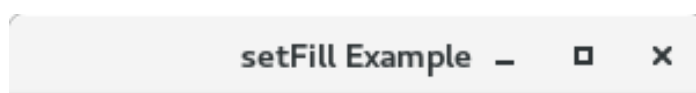
6.7: JavaFX Line

In general, Line can be defined as the geometrical structure which joins two point s (X1,Y1) and (X2,Y2) in a X-Y coordinate plane. JavaFX allows the developers to create the line on the GUI of a JavaFX application. JavaFX library provides the class **Line** which is the part of **javafx.scene.shape** package.

How to create a Line?
Follow the following instructions to create a Line.

- Instantiate the class **javafx.scene.shape.Line**.
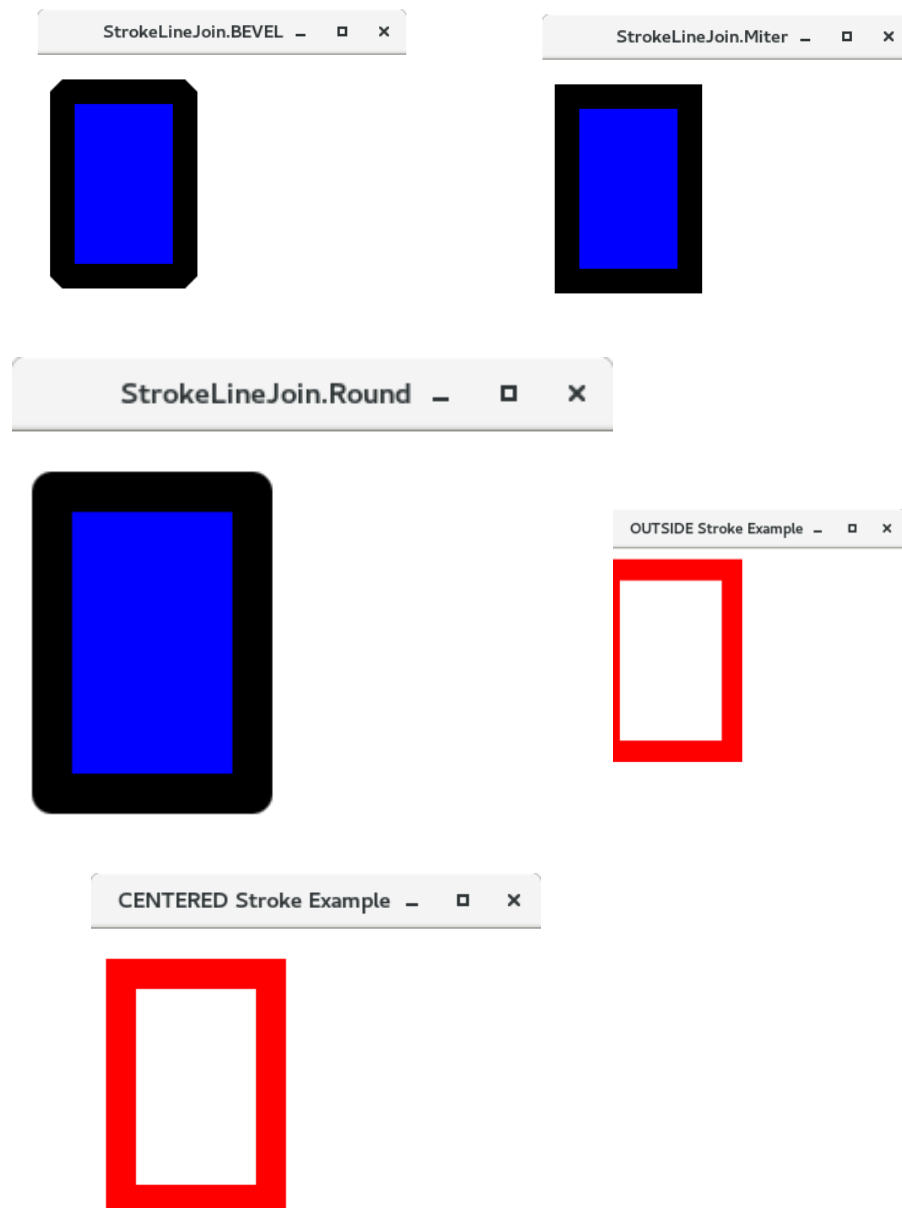
- set the required properties of the class object.
- Add class object to the group

Properties

Line class contains various properties described below.

Property	Description	Setter Methods
endX	The X coordinate of the end point of the line	setEndX(Double)
endY	The y coordinate of the end point of the line	setEndY(Double)
startX	The x coordinate of the starting point of the line	setStartX(Double)
startY	The y coordinate of the starting point of the line	setStartY(Double)

Example 1:

```
package application;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.Group;
import javafx.scene.shape.Line;
import javafx.stage.Stage;
public      class  LineDrawingExamples extends Application{

  @Override
  public      void     start(Stage primaryStage) throws Exception {
    // TODO Auto-generated method stub
    Line line = new      Line(); //instantiating Line class
    line.setStartX(0); //setting starting X point  of Line
    line.setStartY(0); //setting starting Y point  of Line
    line.setEndX(100); //setting ending X point  of Line
```

```
line.setEndY(200); //setting ending Y point  of Line
Group root = new      Group(); //Creating a Group
root.getChildren().add(line);  //adding  the  class    object //to  the
group
Scene scene = new      Scene(root,300,300);
primaryStage.setScene(scene);
primaryStage.setTitle("Line Example");
primaryStage.show();


   }
   public      static void     main(String  [] args) {
      launch(args);
   }


}
```

Output:
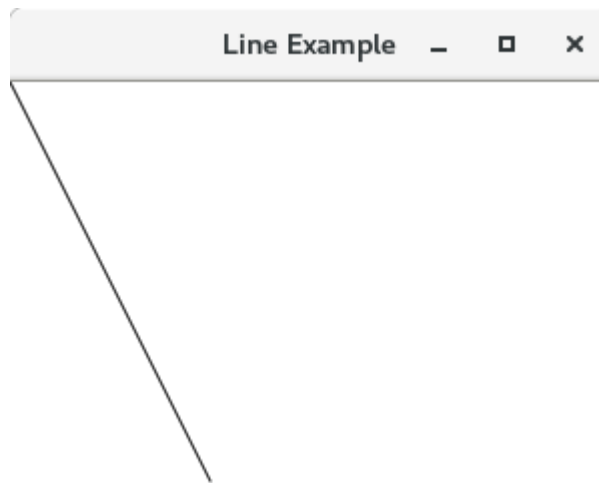
Example 2 : Creating Multiple Lines

```
package application;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint .Color;
import javafx.scene.shape.Line;
import javafx.stage.Stage;
public      class  LineDrawingExamples extends Application{
```

```
public      static void    main(String  [] args) {
   launch(args);
}
@Override
public      void    start(Stage primaryStage) throws Exception {
   // TODO Auto-generated method stub
   primaryStage.setTitle("Line Drawing Examples");
   Line line1 = new      Line(10,50,150,50);
//Line(startX,startY,endX,endY)
   Line line2 = new      Line(10,100,150,100);
   Line line3 = new      Line(10,50,10,100);
   Line line4 = new      Line(150,50,150,100);
   Group root = new      Group();
   root.getChildren().addAll(line1,line2,line3,line4);
   Scene scene = new      Scene (root,300,200,Color.GREEN);
   primaryStage.setScene(scene);
   primaryStage.show();
   }


}
```

Output:
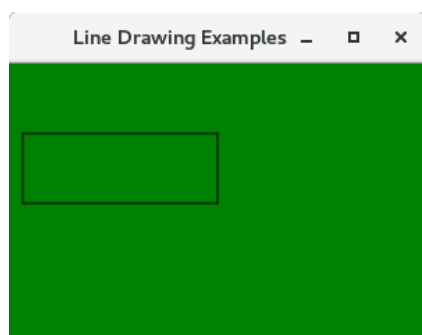
6.8: JavaFX Circle

A circle is a special type of ellipse with both of the focal point s at the same position. Its horizontal radius is equal to its vertical radius. JavaFX allows us to create Circle on the GUI of any application by just instantiating **javafx.scene.shape.Circle** class . Just set the class properties by using the instance setter methods and add the class object to the Group.

Properties

The class properties along with the setter methods and their description are given below in the table.

Property	Description	Setter Methods
centerX	X coordinate of the centre of circle	setCenterX(Double value)
centerY	Y coordinate of the centre of circle	setCenterY(Double value)
radious	Radius of the circle	setRadius(Double value)

Example:

```
package application;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint .Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
public      class  Shape_Example extends Application{

  @Override
  public      void     start(Stage primaryStage) throws Exception {
    // TODO Auto-generated method stub
  primaryStage.setTitle("Circle Example");
  Group group = new      Group();
  Circle circle = new      Circle();
  circle.setCenterX(200);
  circle.setCenterY(200);
  circle.setRadius(100);
```

```
circle.setFill(Color.RED);
group.getChildren().addAll(circle);
Scene scene = new        Scene(group,400,500,Color.GRAY);
primaryStage.setScene(scene);
primaryStage.show();
}
public      static void     main(String  [] args) {
launch(args);
}

}
```
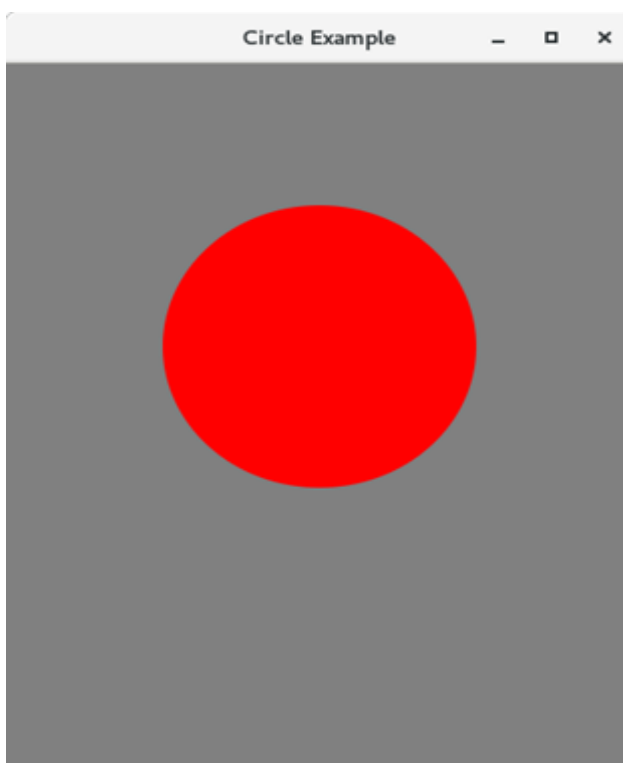

6.9: JavaFX Graphical Effects and Transformations

As one of the most powerful GUI toolkits to build the rich client application, JavaFX gives us a very handy set of graphical effects and transformations that make it possible to give more visual effects and int eractivity to the user int erface. These features are important for the development of modern and int eractive applications that catch the user's eye. While the graphical effects allow you to apply visual changes to nodes (like blur, drop shadows, and coloring) the transformations allow you to modify the geometrical properties of

nodes like scale, rotation and translation. These tools are very important to understand and need to use thoroughly in order to develop rich user int erface-based applications for JavaFX developer. JavaFX effects are essentially visual transformations that change how a node is rendered while keeping the node's underlying geometry and layout int act. For example, you may use a Gaussian blur to smooth the edges of an image or add a drop shadow to give some depth. Transformations edit the position, size, or orientation of the node within the scene graph, in contrast. You could scale the button to make it grow or shrink, rotate the label to write it in an angle, or translate the image to drag it across the screen. These transformations are non-destructive, meaning the node's original properties remain unchanged. JavaFX comes with many built-in effects and transformations, all with their own parameters and options. This capability enables developers to deliver an expansive range of visual tweaks, from subtle touches to bold transformations. For instance: A developer could create a night mode effect using a color adjust effect to invert the color scheme of their int erview application, or add a reflection effect to their app's button to make it shiny. All these effects and transformations could be animated and give you a very nice dynamic visual experience. The Hierarchical structure of the elements that minimal JavaFX Scene Graph reflected onto JavaFX animation philosophyEffects and Transformations The effects are applied to the specific nodes, the transformations change the node and all children elements. Because of this hierarchical nature multiple effects and transformation could be done to different nodes in the scene graph resulting in complex visual effects. In addition, JavaFX is hardware accelerated for effects and transformations, meaning that they will be rendered efficiently and smoothly even for complex scenes. Hardware acceleration is especially crucial in scenarios involving animations and int eractive applications, where performance takes center stage. For example, if a developer wants to design an eye-catching button that increases in size as the user hovers over it. They would use a scale transform on the button and an animation toward scale factor using a timeline. In the same vein, a developer may create a drop shadow effect to highlight a selected item in a list view as visual acknowledgment of user int eraction. Effects and transformations are naturally int egrated in JavaFX, making it easier to produce visually stunning applications with minimal coding effort.

Check Your Progress

1. **What is the "Scene Graph" in JavaFX?**

   ……………………………………………………………………
   ……………………………………………………………………
   ……………………………………………………………………
   ……………………………………………………………………

2. **What class would you use to create a solid red color?**

   ……………………………………………………………………
   ……………………………………………………………………
   ……………………………………………………………………
   ……………………………………………………………………

6.10: Summary

This unit covered the building blocks of 2D graphics in JavaFX. We learned that all visual elements are part of a **Scene Graph**, a hierarchical tree of nodes. The javafx.scene.shape package provides numerous classes for creating 2D shapes, including Line, Rectangle, Circle, and Polygon. We explored how to control the appearance of these shapes by setting their fill and stroke properties. The fill is defined by a Paint object, which can be a solid Color or a more complex gradient. The stroke controls the appearance of the shape's outline, including its width and color. Finally, we saw how the Text class is used to display and style text, allowing control over its font, size, and content.

6.11: Exercises

Multiple Choice Questions

1. Which class is used to create a simple, filled, solid-color circle?
   a) Circle
   b) Ellipse
   c) Arc
   d) Shape
   Answer: a) Circle

2. To set the outline color of a shape, you would modify its:
   a) fill
   b) stroke

c) color

d) border

Answer: b) stroke

3. Which class is the base class for all 2D shapes in JavaFX?

a) Node

b) Shape

c) Graphics

d) Canvas

Answer: b) Shape

4. To display the text "Hello World" in your scene, you would typically use the:

a) Label class

b) String class

c) Text class

d) TextField class

Answer: c) Text class

5. The Paint class is the superclass for:

a) Color and Gradient

b) Stroke and Fill

c) Shape and Text

d) Node and Scene

Answer: a) Color and Gradient

Descriptive Questions:

1. What is a Scene Graph? Explain its tree-like structure and the difference between a "branch node" (like Group or Pane) and a "leaf node" (like Rectangle or Text).

2. Write a JavaFX code snippet that creates a blue Rectangle with a 3-pixel-wide black border (stroke).

3. How do you create a linear gradient fill in JavaFX? Describe the key parameters, such as start/end points and color stops.

4. Explain the difference between the Text node and the Label control. When would you use one over the other?

5. What is the difference between setting a shape's fill to Color.TRANSPARENT and setting it to null?

6.12: References and Suggested Readings

- Dea, J. et al. (2014). *JavaFX 8: Introduction by Example*. Apress.
- Liang, Y. D. (2019). *Introduction to Java Programming*. Pearson.
- Oracle. (2025). *JavaFX API Documentation*

Block 2: **Java FX Technology**

Unit 7: Java FX Effects

Structure

7.1: Introduction

JavaFX provides a powerful set of **visual effects** that can be easily applied to any node in the scene graph (like shapes, text, or even layout panes). These effects allow you to add a professional polish to your application without complex graphics programming. The javafx.scene.effect package contains a wide range of effects, including **drop shadows** to create a sense of depth, **blurs** to soften or obscure elements, **reflections** to create a "wet floor" look, and **color adjustments** like bloom, sepia tone, and lighting. This unit explores how to instantiate these effect objects and apply them to nodes to dramatically enhance the visual quality of your user interface.

7.2: Learning Outcomes

After completing this unit, you will be able to:

1. Explain the purpose of the javafx.scene.effect package.
2. Apply a simple effect (like DropShadow or GaussianBlur) to a JavaFX node.
3. Understand that effects can be "chained" together to create more complex results.
4. Identify several common effects, such as Reflection, Lighting, Glow, and SepiaTone.
5. Modify the properties of an effect (e.g., changing the radius of a blur or the offset of a shadow).

7.3: Java FX Effects

JavaFX supports numerous graphics effects out of the box: notably blur, drop shadow, color adjustment, and reflection. These effects can

then be triggered on any node in the scene graph to provide an application increased visual fidelity. Now, let us show some of these effects with working code in Java. The first mentioned new effect is the GaussianBlur effect new ly add which is blurring the contents of a node. This is used for illusion of 2D or physical emphasis. Here's a simple example:

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.paint .Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.scene.effect.GaussianBlur;

public class GaussianBlurExample extends Application {
    @Override
    public void start(Stage primaryStage) {
        Rectangle rect = new Rectangle(200, 100, Color.BLUE);
        GaussianBlur blur = new GaussianBlur();
        blur.setRadius(10); // Adjust the blur radius
        rect.setEffect(blur);

        StackPane root = new StackPane(rect);
        Scene scene = new Scene(root, 400, 200);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Gaussian Blur Example");
        primaryStage.show();
    }

    public static void main(String [] args) {
        launch(args);
    }
}
```

In the example, we create a Rectangle and then apply a GaussianBlur effect to it. The setRadius() method defines the amount of blur. [Next] The DropShadow effect creates a shadow behind a node to help emulate depth. Here's an example:

Java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.paint .Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.scene.effect.DropShadow;

public       class  DropShadowExample extends Application {
   @Override
   public      void     start(Stage primaryStage) {
      Circle circle = new       Circle(50, Color.RED);
      DropShadow shadow = new       DropShadow();
      shadow.setRadius(20);
      shadow.setColor(Color.BLACK);
      circle.setEffect(shadow);

      StackPane root = new       StackPane(circle);
      Scene scene = new       Scene(root, 200, 200);
      primaryStage.setScene(scene);
      primaryStage.setTitle("Drop Shadow Example");
      primaryStage.show();
   }

   public      static void    main(String  [] args) {
      launch(args);
   }
}
```

There, a Circle is defined and the DropShadow effect is used. Methods setRadius() and setColor() governs the shadow appearance. Using ColorAdjust effect This allows you to modify node's hue and saturation, brightness and contrast. This allows for potential color variations or special effects.

Java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
```

```java
import javafx.scene.paint .Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.scene.effect.ColorAdjust;

public     class  ColorAdjustExample extends Application {
  @Override
  public    void    start(Stage primaryStage) {
    Rectangle rect = new     Rectangle(200, 100, Color.GREEN);
    ColorAdjust adjust = new     ColorAdjust();
    adjust.setHue(0.2);
    adjust.setSaturation(0.5);
    rect.setEffect(adjust);

    StackPane root = new     StackPane(rect);
    Scene scene = new     Scene(root, 400, 200);
    primaryStage.setScene(scene);
    primaryStage.setTitle("Color Adjust Example");
    primaryStage.show();
  }

  public    static void    main(String  [] args) {
    launch(args);
  }
}
```

In this example, the ColorAdjust effect is used to define a color on a Rectangle. The color value is controlled through the setHue() and setSaturation() methods. Next, we have the Reflection effect that ensures what you see in the node above it, is also seen right below it, providing it a mirror kind of effect.

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.StackPane;
import javafx.scene.control.Label;
import javafx.stage.Stage;
import javafx.scene.effect.Reflection;

public     class  ReflectionExample extends Application {
```

```
@Override
public      void      start(Stage primaryStage) {
    Label label = new      Label("Reflection");
    Reflection reflection = new      Reflection();
    reflection.setFraction(0.7); // Adjust the reflection fraction
    label.setEffect(reflection);

    StackPane root = new      StackPane(label);
    Scene scene = new      Scene(root, 200, 100);
    primaryStage.setScene(scene);
    primaryStage.setTitle("Reflection Example");
    primaryStage.show();
}


public      static void    main(String  [] args) {
    launch(args);
}
}
```

In this example, a Reflection is applied to a Label. The length of the reflection is controlled with the setFraction() method. All of these samples show you how to use graphical effects in JavaFX. Developers can use a combination of these effects by adjusting their properties to produce a variety of visual enhancements.

Check Your Progress

1. **What is the main purpose of the javafx.scene.effect package?**

   ………………………………………………………………………
   ………………………………………………………………………
   ………………………………………………………………………
   ………………………………………………………………………

2. **Name two common affects you can apply to a node.**

   ………………………………………………………………………
   ………………………………………………………………………
   ………………………………………………………………………
   ………………………………………………………………………

7.4: Summary

This unit demonstrated the power of the JavaFX effects framework. We learned that visual effects are objects found in the javafx.scene.effect package and can be applied to any scene graph node using its setEffect() method. These effects enhance an application's visual appeal without requiring complex manual graphics rendering. We covered common effects like DropShadow for adding depth, GaussianBlur for softening, and Reflection for creating a mirror image. We also noted that effects are customizable via their properties (like radius, color, offset) and can even be chained together by setting one effect as the input for another, allowing for sophisticated visual combinations.

7.5: Exercises

Multiple Choice Questions

1. Which effect is used to create a "soft edge" or "out of focus" look?

    a) DropShadow

    b) Reflection

    c) GaussianBlur

    d) Glow

Answer: (c) GaussianBlur

2. To add a simple shadow behind a Text node, you would use:

    a) DropShadow

    b) InnerShadow

    c) Lighting

    d) SepiaTone

Answer: (a) DropShadow

3. Which method is called on a node to apply an effect?

    a) setEffect(Effect e)

    b) applyEffect(Effect e)

    c) addEffect(Effect e)

    d) setStyle("-fx-effect: ...")

Answer: (a) setEffect(Effect e)

4. The effect that simulates a "wet floor" mirror image is called:

    a) Mirror

    b) Flip

c) Water

d) Reflection

Answer: (d) Reflection

5. How can you apply two effects (e.g., a Blur and a Shadow) to a single node?

a) It is not possible.

b) Call setEffect() twice.

c) Chain the effects, e.g., shadow.setInput(blur).

d) Add both effects to a List.

Answer: (c) Chain the effects

Descriptive Questions:

1. What is the purpose of the DropShadow effect? Describe its main properties (e.g., offsetX, offsetY, radius, color) and what each one controls.

2. Explain the concept of "chaining" effects. Write a code snippet that applies a GaussianBlur first, and then applies a Glow effect to the *result* of the blur.

3. What is the Lighting effect? Describe its purpose and one of the light sources it can simulate (e.g., Light.Distant or Light.Point).

4. How would you create a "sepia tone" (old-fashioned photo) effect for an ImageView? Which effect class would you use?

5. What is the difference between an InnerShadow and a DropShadow?

7.6: References and Suggested Readings

- Heckler, M. (2014). JavaFX 8: Introduction by Example. Apress.

- DiMarzio, J. F. (2014). JavaFX: A Beginner's Guide. McGraw-Hill Education.

- Vos, J., Grunwald, W., Ioannidis, S., & Chin, G. (2017). Pro JavaFX 9/10/17. Apress.

Block 2: **Java FX Technology**

Unit 8: Java FX Transformations

Structure

8.1: Introduction

Beyond 2D shapes and effects, JavaFX provides a powerful **transformation** framework to change a node's geometry *without* altering the node itself. Transformations allow you to move (translate), rotate, scale (resize), and skew (shear) any node in the scene graph. These operations are fundamental to creating dynamic user int erfaces and animations. For example, you can make a button grow when hovered over (scaling), spin an image around its center (rotation), or slide a component int o view (translation). This unit int roduces the different types of 2D and 3D transformations available in the javafx.scene.transform package and demonstrates how to apply them to nodes.

8.2: Learning Outcomes

After completing this unit, you will be able to:

1.   Define "transformation" and explain why it's useful in a GUI.

2.   Apply basic 2D transformations to a node: Translate, Rotate, Scale, and Shear.

3.   Understand that transformations can be combined.

4.   Differentiate between a 2D transformation and a 3D transformation (like Rotate around the X or Y axis).

5.   Specify the "pivot point " for transformations like rotation and scaling.

8.3: JavaFX Transformations:

This is due to JavaFX transformations, which enables developers to change the spatial features of the nodes, including scaling, rotation and translation. This is a crucial process for building int eractive and responsive user int erfaces. The Scale transformation is used to resize a node. Here's an example:

```java
import javafx.application.Application;
import javafx.scene.layout.Pane;
import javafx.scene.paint .Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.transform.Rotate;
import javafx.scene.transform.Scale;
import javafx.scene.transform.Translate;

public    class  TransformDemo extends Application {
  @Override
  public    void    start(Stage stage) {
    Rectangle rect = new      Rectangle(100, 60,
Color.CORNFLOWERBLUE);
    // Apply transformations
    rect.getTransforms().addAll(
      new      Translate(100, 100),
      new      Rotate(45, 50, 30),
      new      Scale(1.5, 1.5)
    );
    Pane root = new      Pane(rect);
    Scene scene = new      Scene(root, 400, 300);
    stage.setTitle("JavaFX Transformations");
    stage.setScene(scene);
    stage.show();
  }
  public    static void    main(String  [] args) {
    launch(args);
  }
}
```

8.4: JavaFX Rotation

Rotation can be defined as the process of rotating an object by a certain angle θ (theta). In JavaFX, the class **javafx.scene.transform.Rotate** represents the Rotation transform.

The image illustrates the rotation transform. the rectangle shown in the image is rotated along the Y-axis by the angle θ. The coordinates of the rectangle gets changed due to the rotation while the edges remains of the same length.
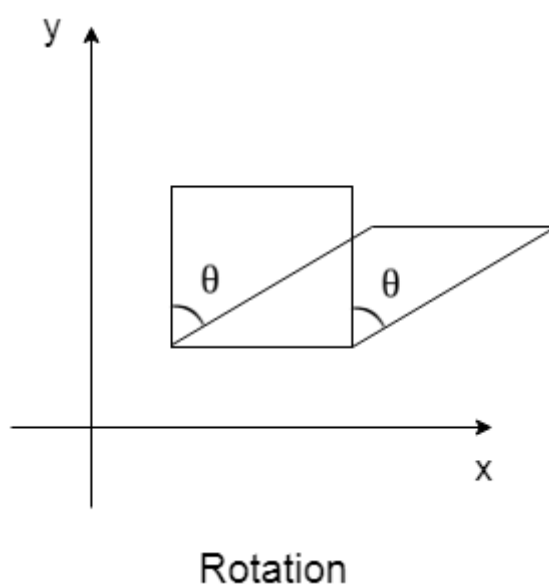
Rotation

Figure 8.1 Rotation

Properties

The properties of the class along with the setter methods are described in the following table.

Property	Description	Setter Methods
angle	It is a double type property. It represents the angle of rotation in degrees.	setAngle(double value)

axis	It is a object type property. It represents the axis of rotation.	setAxis(Point 3D value)
pivotX	It is a double type property. It represents the X coordinate of rotation pivot point .	setPivotX(double value)
pivotY	It is a double type property. It represents the Y coordinate of rotation pivot point .	setPivotY(double value)
pivotZ	It is a double type property. It represents the Z coordinate of rotation pivot point .	setPivotZ(double value)

Constructors

The class contains six constructors.

1. **public   Rotate() :** creates the rotate transform with the default parameters.

2. **public   Rotate(double angle) :** creates the rotate transform with the specified angle measured in degrees. Pivot point s are set to (0,0).

3. **public   Rotate(double angle, Point 3D axis) :** creates the 3D rotate transform with the specified transform. Pivot point s are set to (0,0,0).

4. **public   Rotate(double angle, double pivotX, double pivotY) :** creates the Rotate transform with the specified angle and pivot coordinate (x,y).

5. **public   Rotate(double angle, double pivotX, double pivotY, double pivotZ) :** creates the Rotate transform with the specified angle and 3D pivot coordinate (x,y,z).

6. **public   Rotate(double angle, double pivotX, double pivotY, double pivotZ,Point 3D Axis) :** creates a 3D Rotate transform with the specified angle and pivot coordinate (x,y,z).

Example:

The following example illustrates the implementation of Rotation transform. Here, we have created two rectangles. One is filled with the lime-green color while the other is filled with the dark-grey color. The dark-grey rectangle is rotated with the angle 30 degree along the pivot point coordinates (100,300).

```
Package application;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint .Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.transform.Rotate;
import javafx.stage.Stage;
public       class  RotateExample extends Application{
@Override
public       void     start(Stage primaryStage) throws Exception {
  // TODO Auto-generated method stub
  // creating the rectangles
  Rectangle rect1 = new      Rectangle(100,100,200,200);
  Rectangle rect2 = new      Rectangle(100,100,200,200);

  // setting the color and stroke for the Rectangles
  rect1.setFill(Color.LIMEGREEN);
  rect2.setFill(Color.DARKGREY);
```

```java
rect1.setStroke(Color.BLACK);
rect2.setStroke(Color.BLACK);

// instantiating the Rotate class .
Rotate rotate = new     Rotate();

//setting properties for the rotate object.
Rotate.setAngle(30);
rotate.setPivotX(100);
rotate.setPivotY(300);

//rotating the 2nd rectangle.
Rect2.getTransforms().add(rotate);

Group root = new     Group();
root.getChildren().addAll(rect1,rect2);
Scene scene = new     Scene(root,500,420);
primaryStage.setScene(scene);
primaryStage.setTitle("Rotation Example");
primaryStage.show();
}
public     static void     main(String  [] args) {
    launch(args);
}
}
```
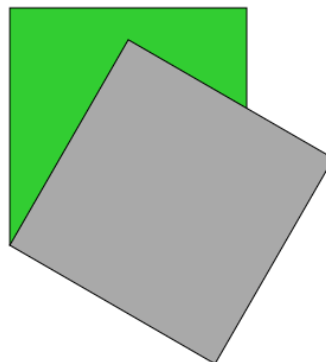
Check Your Progress

1. **What transformation would you use to move a node 50 pixels to the right?**

………………………………………………………………
………………………………………………………………
………………………………………………………………
………………………………………………………………

2. **What transformation would you use to make a node twice as large?**

………………………………………………………………
………………………………………………………………
………………………………………………………………
………………………………………………………………

8.5: Summary

This unit covered JavaFX transformations, which are operations that modify a node's coordinate space. We learned that transformations are non-destructive, meaning they don't change the node's actual properties, only how it is rendered. The primary 2D transformations are Translate (to move), Rotate (to spin), Scale (to resize), and Shear (to skew). These transformations are objects from the javafx.scene.transform package and are applied to a node's getTransforms() list. We also noted the importance of the pivot point , which defines the center for rotation and scaling operations. Finally, we recognized that JavaFX also supports 3D transformations, allowing for complex rotations in 3D space.

8.6: Exercises

Multiple Choice Questions

1. Which transformation is used to move a node's position?
   a) Scale
   b) Rotate
   c) Translate
   d) Shear

   Answer: (c) Translate

2. To make a node appear twice as wide and half as tall, you would use:

   a) Scale(2.0, 0.5)

   b) Scale(2.0, -0.5)

   c) Shear(2.0, 0.5)

   d) Translate(2.0, 0.5)

Answer: (a) Scale(2.0, 0.5)

3. Which transformation is used to "spin" a node around a point ?

   a) Shear

   b) Rotate

   c) Spin

   d) Translate

Answer: (b) Rotate

4. Transformations are applied to a node by adding them to its:

   a) Effect list

   b) Children list

   c) Transforms list

   d) Style property

Answer: (c) Transforms list

5. The "pivot point " is most relevant for which two transformations?

   a) Translate and Shear

   b) Rotate and Scale

   c) Translate and Scale

   d) Shear and Rotate

Answer: (b) Rotate and Scale

Descriptive Questions:

1. What is the key difference between modifying a node's layoutX/layoutY properties and applying a Translate transformation?

2. Write a JavaFX code snippet that creates a Rectangle and applies a Rotate transformation to spin it 45 degrees around its own center.

3. What is a Shear transformation? Provide a real-world analogy for what it does to a shape.

Notes

4. How can you apply multiple transformations to a single node? Show a code example that first scales a node and then translates it.

5. Explain the purpose of the pivot point properties (pivotX, pivotY) in a Rotate transformation. What happens if you don't set them?

8.7: References and Suggested Readings

- Schildt, H. (2018). *Java: The complete reference* (10th ed.). McGraw-Hill Education.
  (Chapter on JavaFX Transformations)
- Chin, S., Vos, J., Grunwald, W., & Weaver, J. (2018). *Pro JavaFX 10*. Apress.
- Oracle. (n.d.). *JavaFX tutorials: Transformations*. Oracle Documentation.
- Oracle. (n.d.). *JavaFX CSS reference guide*. Oracle Documentation.

Block 2: **Java FX Technology**

9: Java FX Animation

Structure

9.1: Introduction

Static int erfaces are functional, but **animation** brings them to life. The JavaFX animation framework, found in the javafx.animation package, provides a simple yet powerful way to create smooth transitions and dynamic effects. Instead of you manually managing a loop to change a property over time, JavaFX's animation engine does all the hard work. This unit focuses on the core of this system: **Transitions**. Transitions allow you to animate a specific property of a node, such as its opacity (FadeTransition), position (TranslateTransition),size (ScaleTransition), or color (FillTransition). We will explore how to define these transitions, set their duration, and control their playback.

9.2: Learning Outcomes

After completing this unit, you will be in a position to:

1. Explain the role of animation in a user int erface.

2. Understand the concept of a Transition as the core animation class .

3. Implement common transitions like FadeTransition, TranslateTransition, RotateTransition, and ScaleTransition.

4. Control animation properties such as duration, autoReverse, and cycleCount.

5. Combine multiple animations to run in parallel (ParallelTransition) or in sequence (SequentialTransition).

9.3: Java FX Animation

JavaFX is a robust and flexible toolkit for creating rich client applications that go beyond static UIs to include dynamic animations, immersive 3D graphics, complex layout management, and a full set of UI controls. JavaFX is highly visual and you can use timelines, transitions, and keyframes to do animation. A Timeline, the heart of JavaFX animation, a time-based driver that fires events at specific periods in time. Transitions, such as FadeTransition, TranslateTransition, and RotateTransition, offer pre-defined animation effects that can be applied to nodes to create animations with less complexity. Keyframes we use instead represent the state of node at a certain point in time, allowing for complex animations to be created by int erpolating between the two states. UI transition animations are smaller snippets of animations and can be utilized to indicate when the user hovers/clicks on UI controls, for example, using a FadeTransition to gradually change the opacity of a button, reducing its visual presence as the button is being changed on hover. For instance, a TranslateTransition can move a shape across the screen, giving it the illusion of motion, and a RotateTransition can rotate an object, energizing a UI. Keyframes Demonstration — Bouncing Ball Animation A composite animation can be created by adding keyframes that change the ball's position and velocity with time, resulting in a more realistic bounce. When used well, animation—especially int eractivity—is useful in JavaFX, but it can also help create a good user experience by giving feedback, highlighting, and improving int erface expressiveness, which makes it more engaging and easy to use. With these animation techniques, developers can add vitality to their applications, infuse energy int o their static int erfaces and engage their user through dynamic experiences. With the ability to orchestrate visual components in time, it enables developers to craft rich user experiences, hence why JavaFX is such a powerful framework for creating progressive, stunning applications.

9.4: Java FX 3D Shapes

With a solid 3D graphics support, JavaFX is capable of rendering rich and int eractive 3D worlds, beyond just 2D int erfaces. To deal with JavaFX 3D shapes one can use class es such as Box, Sphere, Cylinder, and MeshView which are representing basic 3D primitives. For example, a Box is used to make a cube or rectangular prism, and a Sphere is an object shaped like a sphere. A Cylinder — as the name signifies — is a cylindrical shape. More advanced 3D models can be made through MeshView, where more advanced details can be created such as vertices, faces, and texture coordinates can be defined. Through Translate, Rotate and Scale properties, the primitives are manipulatable in 3D space, meaning we can position the geometry, rotate its position and also scale. JavaFX also has lighting and material properties to make 3D scenes more realistic. As you shine light sources (Point Light, AmbientLight, …) on a scene, the 3D objects display shadows and highlights, which gives your objects a sense of depth. Materials (e.g. PhongMaterial) specify the surface char acteristics of the 3D object and govern its color, reflectivity, and texture. These properties can be used to create 3D scenes that never fail to look great, now on par with things you would expect to see built with full 3D graphics libraries. A PhongMaterial, for instance, can be added to a Sphere to make the shape appear metallic or glossy, or multiple Point Lights can be added to the scene to get realistic-looking lighting effects. JavaFX also supports accurate 3D models generated in external modeling programs like Blender or Maya, using the OBJ and FBX file formats for importing. By supporting these new formats, this allows developers to use high-fidelity 3D assets in their own applications, and breaks open new avenues for 3D experiences. This gives developers complete control to manipulate 3D shapes, lights, and materials to produce visually appealing applications ranging from int eractive 3D visualizations to rich gaming experiences, showcasing the platform's versatility and capability to manage advanced graphics.

9.5: Java FX Layout

Having good layout management is an essential part of building functional and visually appealing user int erfaces. The main layout panes provided by JavaFX are BorderPane, HBox, VBox, GridPane and StackPane, each int ended to layout UI components in a particular

fashion. Another example is BorderPane, which divides the layout int o top, bottom, left, right, and center sections, allowing you to create a structured layout with different sections. HBox and VBox – Helpful when needing to arrange components on a single line, horizontally or vertically GridPane: A componen that arranges UI in a grid, giving accurate control over each UI components position and alignment For example, StackPane, which stacks Nodes on top of each other. These layout panes can contain other panes to create int ricate and adaptable layouts. As an example, we can structure the application's overall layout with a BorderPane, the top region with an HBox for a toolbar, the left region with a VBox for a navigation menu, and the center region with a GridPane for a data entry form. VBox, StackPane, etc., depending on the expected behavior, and they can also use layout properties on their own (e.g., alignment, padding, and spacing) to adjust the components' appearance and behavior. The alignment properties determine the placement of UI components in relation to their parent container, and padding and spacing properties add visual distance between UI components and the parent container or between neighboring elements. What is more, JavaFX support CSS styling which allows developers to style the layout panes and UI components according to their own custom style guide, providing better project visual consistency and aesthetics. Static ImportsIn many cases, including the libraries you need is sufficient to get you started, but if you want more control over your final distribution, there are some additional steps you can take to reduce the amount of unused code from your bundles. Developers can learn these layout techniques to make a very int uitive, responsive, and good-looking application to have a good user experience. JavaFX layout management is flexible and powerful, treason to build modern several applications.

9.6: Java FX UI Controls

JavaFX visuals are a set of controls that includes buttons, text fields, labels, checkboxes, radiobuttons, and combo boxes. On the other hand, buttons are int ended for actions, for example in submitting a form and navigating to a new screen. Text fields are used for receiving user inputs and for displaying text that can be entered and altered by users. Labels therefore are static text that helps inform the user about what is required. Checkboxes, radio buttons and combo boxes are used to select

options. New UI control is automatically assigned with set of properties/methods that could be used to customize its appearance and behavior. These attributes provide char acteristics for certain types of controls— for instance, a button's text, font, and color can be changed, and a text field can have its prompt text or input validation configured. JavaFX is a rich user int erface toolkit for Java apps. For example, you can bind an event handler to a button that allows the button to perform one or more action(s) when it is clicked. Implemented in the form of UI controls, they can be styled with CSS and designed. Custom styles enable you to create buttons with a flat or gradient appearance, or adjust how text fields appear with rounded corners or a custom border. In addition, JavaFX offers several UI controls tailored for tabular and hierarchical data, including TableView and TreeView respectively. TableView display data in a table format, in columns and rows TreeView display data, in a tree structure, in parent and child nodes The data consumer application mentioned above needs these specialized UI controls to render the data places mentioned above and to manipulate these complex data to accomplish the goal. JavaFX also offers a wide variety of UI controls that developers can use to create highly int eractive and visually appealing applications. These UI controls and event handling mechanisms contribute to a rich user experience, the art of making applications that are functional yet engaging is a domain for you to discover. This is why JavaFX UI controls will always be a great toolkit to use for developing modern int eractive applications.

9.7: JavaFX Images

JavaFX has various components that can be used in tandem such as images and event handling which allows us to create dynamic and int eractive UIs. Combining these two flavours of software provides developers with the power to construct applications that not only present aesthetically pleasing content, but also respond dynamically with int elligence in accordance to user input. Let's take an example, say an application that showcases a gallery of pictures. (Users can browse the gallery by hitting navigation buttons or swiping on the screen.) We also define TextView3 and TextView4 objects for our UI; these will be used to display the information about the image and when buttons are pressed (Gallery contains images) each image can be represented by ImageView object, and our Next and Previous buttons

will be represented by Button objects. The navigation buttons and the ImageView objects can have event handlers that respond to user clicks and touch gestures. If the user clicks on a navigation button, the event handler may change the contents of the ImageView to the next or last picture from the gallery. When the user swipes on the screen, the event handler can detect the swipe gesture and update the ImageView accordingly. JavaFX provides drag-and-drop, so users can drag images around the application. You can do this by using the setOnMousePressed(), setOnMouseDragged(), setOnMouseReleased() methods of the ImageView class . When the user click the mouse button on the ImageView, the setOnMousePressed() event handler is executed and it would be possible to record the initial position of the mouse point er. The position of the ImageView (the one to be dragged) can be updated based on mouse movement in the setOnMouseDragged() event handler when the user drags the mouse. The setOnMouseReleased() event handler can be used to finalize the drag-and-drop operation when the user releases the mouse button.

9.8: JavaFX Event Handling

JavaFX Application Lifecycle and Event Handling JavaFX Application Lifecycle And Event Handling JavaFX allows developers to develop int eractive In this article above things will be more clear, as JavaFX provides a mechanism for working with images. Image loading and manipulation is important for dynamic and int eractive applications. JavaFX offers comprehensive support for managing multiple image formats such as PNG, JPEG, and GIF, using the javafx. scene. image. Image class . This container provided by RwImage gives the flexibility to load images from a multitude of sources, including local files, URLs, or input streams. The caption for the progress of loading an image is to create an Image object and point to an image source. For example, an image can be loaded from a local file by using the image constructor and passing the file path as an argument. Likewise, for an image, the URL String is also passed to the constructor for loading an image from a URL. After the creation of Image object, it can be rendered inside the JavaFX stage using the javafx. scene. image. ImageView class . The ImageView serves as a node to draw the image in the scene graph. Developers can use the setImage() method to assign the Image object to the ImageView. In addition to just displaying

them, JavaFX provides many ways to deal with images. The ImageView class has methods like setFitWidth() and setFitHeight() to scale the image to fit the provided dimensions. By default, images are scaled proportionally, so setPreserveRatio() can be used to preserve the aspect ratio of an image to avoid distortion. Using the getTransforms() method of the ImageView class, developers can also apply in-depth transformation on image like rotation, translation, scaling, etc. This will return an observable list of Transform objects which can be modified as necessary to produce the desired visual effects. To rotate an image, for example, add a Rotate transform to the list, indicating the rotation angle. For JavaFX you can perform image filtering which enables a developer to apply an image with different effects like Blur, color shading, drop shadow, etc. These effects can be applied using the setEffect() method of the ImageView class. For example, to create a blur effect, a GaussianBlur effect can be instantiated and assigned to the ImageView. JavaFX also has low-level image manipulation class es in its pixelreader and pixelwriter methods.

Check Your Progress

1. **What is the purpose of a Transition in JavaFX?**

……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………

2. **What class would you use to make a node fade in and out?**

……………………………………………………………………
……………………………………………………………………
……………………………………………………………………
……………………………………………………………………

9.9: Summary

This unit int roduced the JavaFX animation framework, a high-level API for creating dynamic user int erfaces. We learned that the easiest way to create animations is by using pre-built Transition class es. These class es, such as FadeTransition, TranslateTransition, RotateTransition, and ScaleTransition, are designed to animate a specific property of a node over a set Duration. We saw how to control animations by setting

their cycleCount (for repetition) and autoReverse (for reversing direction). Finally, we explored how to orchestrate complex animations by using SequentialTransition to play animations one after another, and ParallelTransition to play multiple animations at the same time.

9.10: Exercises

Multiple Choice Questions

1. Which class is used to make a node gradually disappear or appear?
   a) TranslateTransition
   b) FadeTransition
   c) ScaleTransition
   d) FillTransition

Answer: (b) FadeTransition

2. To make an animation repeat 5 times, you would set its:
   a) duration
   b) cycleCount
   c) autoReverse
   d) repeat

Answer: (b) cycleCount

3. Which class would you use to play two animations at the exact same time?
   a) SequentialTransition
   b) ParallelTransition
   c) AnimationTimer
   d) TransitionGroup

Answer: (b) ParallelTransition

4. To move a shape from one side of the screen to the other, the best choice would be:
   a) RotateTransition
   b) ScaleTransition
   c) TranslateTransition
   d) PathTransition

Answer: (c) TranslateTransition

5. To make an animation play forward and then backward, you would set:
   a) autoReverse to true and cycleCount to 2
   b) autoReverse to true

c) cycleCount to -1

d) autoReverse to false

Answer: (b) autoReverse to true

Descriptive Questions:

1. What is the core difference between a Timeline animation and a Transition? When would you use one over the other?

2. Write a JavaFX code snippet that makes a Circle continuously fade in and out (indefinitely).

3. Explain the purpose of SequentialTransition and ParallelTransition. Provide a simple example of where you would use each.

4. What is an "int erpolator" (e.g., Int erpolator.EASE_IN) in the context of JavaFX animation? How does it affect an animation?

5. How would you create an animation that moves a Rectangle in a perfect circle? (Hint : You may need PathTransition).

9.11: References and Suggested Readings

- Schildt, H. (2018). *Java: The complete reference* (10th ed.). McGraw-Hill Education. (Chapter on JavaFX Animation)

- Chin, S., Vos, J., Grunwald, W., & Weaver, J. (2018). *Pro JavaFX 10*. Apress.

- Oracle. (n.d.). *JavaFX tutorials: Animation*. Oracle Documentation.

- OpenJFX. (n.d.). *Getting started with JavaFX: Animation*. OpenJFX Documentation.

Block 3: Servlet Technology

Unit 10: J2EE Introduction and Architecture

Structure

10.1: Introduction

So far, we have focused on Java Standard Edition (Java SE), which is used for building desktop applications (like JavaFX) and console programs. Now, we move to **Java 2 Platform, Enterprise Edition (J2EE)**, now known as **Jakarta EE**. This is not a different language; it is a *specification* and a set of technologies built *on top* of Java SE designed specifically for building large-scale, multi-tiered, and reliable server-side applications. J2EE provides the architecture for handling complex business logic, managing web traffic, and int eracting with databases. This unit int roduces the core concepts of J2EE, its multi-tier architecture, and the crucial idea of "containers" that manage the life cycle of server-side components.

10.2: Learning Outcomes

After completing this unit, you will be able to:

1. Define J2EE (Jakarta EE) and its purpose.

2. Differentiate between Java SE and J2EE.

3. Describe the J2EE multi-tier architecture (Client, Web, Business, EIS tiers).

4. Explain the concept of a J2EE "container" (e.g., Web Container, EJB Container).

151

5. Identify key J2EE technologies like Servlets, JavaServer Pages (JSP), and Enterprise JavaBeans (EJB).

10.3: J2EE Introduction and Architecture

You are currently reading about Jakarta EE (Formerly J2EE or Java EE) Latest Version: Jakarta EE 10, learn how to use as old J2EE Java Enterprise Edition. In the late 1990s, J2EE was int roduced as a complement to the Java Standard Edition (JSE) to create a standardized framework for enterprise application development, and it was a product of Sun Microsystems. We believed so strongly in a complete int egrated development environment that could solve many-faceted enterprise computing problems without compromising the primary promise of Java "write once run anywhere", that we offered tutorial programs, synergies with upstream partners, and pushed through customer accounts manager having knowledge beyond database and applications servers products. This architectural shift was a significant departure from the monolithic application designs that preceded it in enterprise systems and int o a more modular, component-oriented methodology to meet the needs of an increasingly distributed and componentized environment of business computing. It was not just a technical specification—J2EE democratized enterprise development by providing common patterns, practices and abstractions, helping folks focus on business logic rather than the underlying infrastructure concerns. J2EE defined standard APIs to connect to databases, messaging, transaction management, web services, and more, establishing a platform upon which third-party vendors, open-source projects, and enterprise engineers could build to create a shared community around a common technology stack. Java EE 5, 6, 7, 8, a.k.a Jakarta EE 9+ (various specifications under the Jakarta EE umbrella — it brings together many specifications and broken-down Enterprise/Server components from Java EE). Even with the emergence of alternative frameworks and architectural approaches, the legacy of J2EE endures, underpinning countless mission-critical applications across diverse industries and shaping the principles of modern enterprise development. In this Unit, we will delve int o the architecture, components, and development methodologies of J2EE, unveiling how this groundbreaking platform laid the foundation for

enterprise application development practices that still echo in modern software engineering.
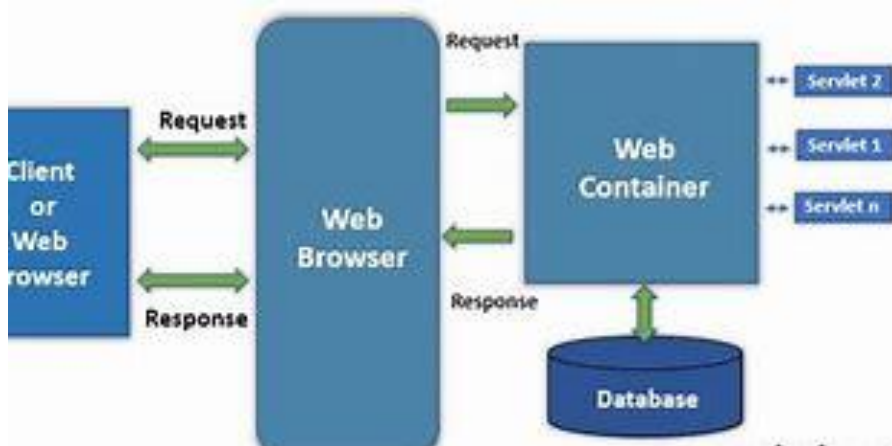
Figure 10.1: Servlet Architecture
[Source: https://th.bing.com/]

Java Enterprise Edition (J2EE) is a specification that leads enterprise application development to be done based on a specification that provides the main int erfaces and the behavior upon which the associations of the applications should be based which multiple vendors can develop compliant implementations. Standardization has played a key role in the emergence of a healthy marketplace of application servers such as IBM WebSphere, Oracle WebLogic, Red Hat JBoss, Apache TomEE, and GlassFish, giving organizations the flexibility of making deployment choices while ensuring application portability. When J2EE was first developed, it was born out of these gaps in the enterprise development world: how to develop distributed systems, common concern over distributed transactions, designing scalable communication protocols and security models. Your fleece-covered IVR is about more than just reducing clicks; it's about reducing payments to outside vendors (those handy-teddies!). The adoption path of J2EE mirrored the class ic technology diffusion curve, with the original adopters being primarily financial services, telecommunications and large scale e-commerce applications, and subsequently expanding int o healthcare, government, manufacturing, and essentially any sector with a significant presence of IT infrastructure. As it has evolved, J2EE has retained fundamental architectural concepts while responding to new models: component-based architecture morphed int o service-oriented architecture, which has moved towards microservices; synchronous communication

models were paired with asynchronous; XML-based configuration was supplanted by configuration by annotation-based methods and convention over configuration; and monolith deployments have crumbled int o both containerized builds and services. Such flexibility has kept J2EE firmly in the conversation, despite massive shifts in development practices. For students as well as practitioners, learning J2EE gives practical expertise in working on enterprise systems and also helps understand architectural patterns that are not technology-bound, which makes the subject an essential cornerstone of any education in enterprise software engineering.

10.4: Fundamental Architecture of J2EE

After reading through Unit 1 of Jeff Lynch's book J2EE made easy, I was left with the impression that the J2EE architecture is simply a multi-tiered distributed application architecture that separates concerns in a way that allows each tier to effectively handle modularity, scalability, and maint ainability issues. J2EE is based on a modified version of the client-server software architecture and is chiefly char acterized by a four-tier architecture consisting of the client tier, web tier, business tier and enterprise information system (EIS) tier. These tiers allow for functional stratification, both logically and physically, so each tier can evolve independently of the other whilst retaining contracted int erfaces for cross-tier conversations. The client tier refers to all user int erface technologies this is where end-users will int eract with the application from web browsers rendering HTML/CSS/JavaScript to native mobile apps, desktop apps using Java Swing or JavaFX, and headless clients like IoT devices or other systems that consume APIs. The web tier is predominantly designed with Servlet and JavaServer Pages (JSP) technologies, this tier accepts HTTP requests, manages user sessions, applies presentation logic, and passes the required data to the business tier and vice versa. The layer separates client implementations from business logic (in this case, a microservice) quite well, which is increasingly common in the contemporary era, allowing for great freedom with how applications are accessed and presented to users. As you know, the business tier, which contains the application's core functionality, business rules, and workflows, is arguably the heart of the J2EE architecture, with such functionality typically being implemented using Enterprise JavaBeans

(EJB). The elements of this tier run in a container environment that manages thing like transaction control, security, concurrency, and lifecycle, so that developers can just think about business logic, not what is under the infrastructure. The third and final tier, the EIS tier, includes the data persistence layer and int egrations with other systems (external systems, external databases, legacy applications, enterprise information systems), and it is accessed through JDBC, JPA, JTA, and JCA technologies. This architectural separation lies at the heart of scalability because each tier can be scaled independently according to the performance needs of that tier and fault tolerance because a problem in one tier is less likely to cascade throughout the entire application. Furthermore, this multi-layer design also enables teams to specialize, making it easier for developers to work on particular segments of the application based on their strengths, whether it be user int erface, business logic, or data handling.

In particular, the container model was one of the more unique architecture innovations int roduced by J2EE, defining a clear separation between infrastructure services and application logic that almost all enterprise development frameworks have followed since. In this paradigm, application components run within specialized runtime environments (also known as containers) that offer standardized services — transaction management, security, resource pooling, lifecycle management, etc — via well-defined contracts instead of through explicit coding. This abbeys the inversion of control pattern, which significantly reduces the amount of boilerplate, adds consistency across apps and enables developers to concentrate mostly on business-specific functionality instead of plumbing. J2EE specifies various container types for particular component models and execution contexts. It is common for web applications to utilize beans, known as Enterprise JavaBeans (EJBs), which are instances of components managed by an EJB container, the runtime environment that manages the lifecycle of an EJB component and its components and creates for an EJB a complex service environment in which xact propagation, instance pooling, and concurrent access to beans x are among the complex services in its remote method invocation. It (web container, or servlet container) serves as the execution environment for Servlets, JSP pages, and other web-tier components, handling request routing, threading models, session management, and HTTP protocol details.

You are supporting and simplifying access to naming, security and remote EJB functionality, rather than J2EE managed component containers, you are offering application client containers against standalone Java-based applications that include J2EE services. Last but not least, we have the applet container which is no longer popular with so many J2EE applications but still loads Java Applets that run inside web browsers. This container-based architecture has the following advantages: it gives you uniform programming models for different app types; it allows the declaration of complex services in terms of deployment descriptors and annotations; it allows components to be reused via standard int erfaces and lifecycles; it allows you to easily impose security on the edges; it allows pooling of resources and instance management for optimization; and it allows deployment flexibility through constant package formats. J2EE framework emphasizes a model of development around the container where you encapsulate functionality in granular well defined, loosely coupled components with well understood responsibilities and int erfaces. Because it steers developers to architectures that are highly cohesive in components and loosely coupled among components in a natural way, these principles can be applied to effective enterprise application design irrespective of technology.

J2EE is itself defined as a building block that comprises other components, services, and APIs to build the platform. Among the finest and most versatile component technologies are Servlets, which extend the functionality of web servers and dynamically builds web content in response to HTTP requests; JavaServer Pages (JSP), which is a template-based component technology for generating dynamic web content, and can separate HTML markup from Java code; Enterprise JavaBeans (EJB), which implements business logic (three varieties exist, including session beans designed to orchestrate business processes, (largely superseded by Java Persistence API) entity beans that represent your data and Message-Driven Beans that implement asynchronous processing; and JavaServer Faces (JSF), which implements a component-based MVC (model-view-controller) framework for web int erfaces. These components are supplemented by the container services of J2EE, which provide cross-cutting capabilities to all components running in the application server environment. They consist of JNDI (Java Naming and Directory Int

erface) for finding resources and components, JTA (Java Transaction API)responsible for transaction management across multiple resources, JAAS (Java Authentication and Authorization Service) for security, JMS (Java Message Service) for reliable asynchronous messages, and JCA (Java Connector Architecture) for int eractions with external enterprise information systems. The platform also includes many specialized APIs that focus on specific enterprise areas: JDBC (Java Database Connectivity) for int eracting with databases; JPA (Java Persistence API) to perform object-relational mapping; JAX-WS and JAX-RS for SOAP and RESTful web services; JavaMail for email; and many other areas that have been added in new er platform versions. Dependency injection is the mechanism by which this rich ecosystem converges around common patterns and practices (starting with JNDI lookup, later formalized around CDI — Contexts and Dependency Injection), and the proliferation of design patterns such as MVC (Model-View-Controller), DAO (Data Access Object), Service Locator, Business Delegate, and Composite Entity. This ecosystem of technologies, services, and patterns culminated in a platform that offers to meet the varied needs of enterprise applications while ensuring uniform maint ainable implementation patterns.

10.5: Evolution and Deployment of J2EE Applications

The platform has matured over time, with each release building upon previous functionality to solve for new enterprise obstacles. On December 12, 1999, the first version of J2EE delivered in the form of the J2EE 1.2 specification, specifying the architecture: Servlet 2.2, JSP 1.1, EJB 1.1 and JDBC 2.0 technologies for standardized enterprise development. J2EE 1.3 brought connector architecture, revamped JMS and EJB 2.0 local int erfaces to this foundation (2001). J2EE 1.4 (2003) brought a crucial direction towards ease of web services int egration, adding JAX-RPC, SOAP with Attachments API for Java (SAAJ), and Java API for XML Registries (JAXR), aligning with the overall industry shift towards service-oriented architectures. The rebranding to Java EE 5, 2006, marked a turning point release in which annotations, dependency injection, and the Java Persistence API combined to significantly reduce the complexity of development, overcoming criticisms of the platform featuring overly verbose frameworks. Java EE 6 (2009): added web profile for lightweight implementations, a

more powerful Contexts and Dependency Injection (CDI) implementation, and built-in support JAX-RS 1.1 for improved RESTful web services. Java EE 7 (2013) added standardized batch processing and concurrency utilities in partnership with updated web technologies including WebSocket and JSON processing. With the release of Java EE 8 (2017), the platform became even more modern — with support for HTTP/2, improved security features, and added support for JSON binding. The move to the Eclipse Foundation resulted in Jakarta EE 9 (2020) which was iterations with primarily the javax namespace adjusted. * to jakarta. *, and Jakarta EE 10 (2022) started to add significant new capabilities under the new governance model. Over the course of this evolution, the platform has exhibited incredible backward compatibility while incrementally moving away from its originally very XML-centric, container-centric model to an increasingly lightweight, annotation-based, developer-centric model— analogous to the broader industry transition from monolithic applications to microservices and cloud-native architectures. But these shifts represent J2EE's ability to evolve with changing paradigms in development while maint aining its core strength: namely, standardization and portability.

A J2EE application goes through a well defined process from its designing, implementation, testing, deployment and maint enance. Architects, for example, break the system requirements down int o the appropriate tiers and components, define boundary int erfaces, data models, and cross-cutting concerns such as security and transaction management (often using UML diagrams, architectural patterns, and J2EE environment reference architectures) during the design time phase. There is also a slice of data focused on the implementation work that typically involves many specialized teams working at the same time: user int erface developers who are creating the JSP pages, Servlets, or JSF components; programmers focused on business logic writing EJBs or CDI beans; data access experts creating JPA entities and repositories; and int egration engineers writing the connectors for external systems. During development, this parallel effort is made possible by J2EE's standardized APIs and component models, which specify clear contracts between different parts of the application. Packaging Modules The build aggregates these varied artifacts int o deployable units according to J2EE's packaging rules: JAR (Java

Archive) files for utility class es and libraries, WAR (Web Application Archive) files for web modules with Servlets and related resources, EJB-JAR files for Enterprise JavaBeans, and EAR (Enterprise Archive) files that bundle multiple modules int o an int egrated application. Arising from the building is deployment, which is the act of installing these packaged artifacts in a J2EE application server that then checks the configuration, satisfies dependencies, sets the right container services and makes the application available to the end-user. DevOps practices are prevalent throughout modern J2EE development, encompassing CI/CD pipelines for the automated execution of build, test and deployment phases; containerization technologies such as Docker, for streamlined environment consistency; orchestration tools such as Kubernetes, for coordinating and scaling deployments; and Infrastructure-as-Code approaches that further replicate deploys through environments. The architecture of J2EE applications is distributed throughout multiple tiers; as a consequence, testing these applications results in a unique set of challenges. J2EE provides significant benefits with this highly standardized approach across its lifecycle as J2EE components become portable (the same application can run on various everywhere implementations), a standard deployment model is applicable across applications regardless of the implementation of the actual application, and common enterprise concerns are addressed using well-defined patterns.

10.6: Key Technologies and Components in J2EE

Servlet technology is the foundation of J2EE's web tier, serving as a Java-centric method for processing HTTP requests and creating dynamic responses in web applications. Servlets are managed in a container that coordinates their lifecycle through specific methods: init() for initialization, service() (usually overridden via doGet(), doPost(), etc.) for request handling, and destroy() for teardown activities. For example, the container takes care of managing the object lifecycle, which means developers don't have to worry about low-level background processing like socket handling, thread management, and protocol details, etc. — they only have to worry about processing the request in an application-specific way. Servlets process incoming requests via HttpServletRequest objects, containing parameters, headers, session info, and request details, and responses via

HttpServletResponse objects, enabling control over content types, headers, status codes, and response content. Servlets provide a performance state—an int erface to manage server-side session maint enance over literate requests through HttpSession int erface, one of the building blocks of web applications. Servlets can be mapped to specific URL patterns by means of deployment descriptors (web. (xml) or annotations (@WebServlet), allowing for flexible routing configurations. In addition to basic request handling, the Servlet API provides features for request dispatching (forwarding or including content from other resources), filtering (int ercepting requests for pre or post-processing), event listeners (receiving notifications about various contextual events such as application startup or session creation), and asynchronous processing (handling long-running operations without blocking threads in the container). Servlet EvolutionThe Servlet specification has evolved hand-in-hand with trends in web development: Servlet 2.5 fitted in annotations to avoid excessive configuration; Servlet 3.0 brought asynchronous processing and programmatic registration; Servlet 3.1 strengthened security and facilitated file uploads; and Servlet 4.0 added HTTP/2 support and server push. At the same time, Servlets remained the underlying technology behind almost all the frameworks in the Java space (JSF, Spring MVC, Struts and other dozens). Servlets serve as reusable components for constructing Java web apps, and while many developers now engage primarily with higher-level abstractions of Servlets, it is critical to understand the underlying fundamentals of Servlets in order to troubleshoot, optimize performance, and deploy your own custom components across the J2EE ecosystem.

The JavaServer Pages (JSP) technology takes the web tier features of J2EE and adds document-centric facilities for generating dynamic content that work naturally in conjunction with the Servlet model. JSP pages consist of standard static (usually HTML markup) and some dynamic tags and embedded Java code, this framework produces a template-based development environment using separate concerns for presentation and business logic. When a JSP page is requested for the first time, the container translates the page int o a Servlet class and compiles that class before executing it, as you would with any Servlet—which means JSP is a syntactic sugar over the Servlet. This process translates standard HTML int o raw text output, JSP directives

() int o package declarations and imports, scriptlets () int o method body code, expressions () int o output statements, declarations () to class - level variables and methods, and different tag types to Java constructs. There are several approaches JSP uses to create dynamic content: scriptlets for embedding raw Java code inside a page, expressions for embedding an evaluated value, the Expression Language (EL) for simplified access to object properties and standard and custom tag libraries for more complex markup-oriented functionalities. The JSP Standard Tag Library (JSTL) includes tags for common tasks such as iteration, condition, XML processing, database access, and i18n, so that embedded Java code can be used much less. Custom tag libraries take this concept further by enabling developers to create re-usable, declarative components that encapsulate domain-specific logic. Over a period of 15 years, JSP technology evolution has proved to be about progressive separation of concerns (JSP 2.0 + Expression Language for easy object access; JSP 2.1 + expression language enhancements with JSF int egration; JSP 2.x line of development to further enhance those while keeping backward compatibility as its guiding principle). Though JSP development has largely been replaced with component-based frameworks such as JavaServer Faces and template engines like Thymeleaf, JSP features still remain in use amongst enterprise applications, especially for their view components via MVC architectures. JSP's sustained relevance can be attributed to its simplified learning curve, natural fit to HTML design flows, its efficient execution model, and seamless compatibility with Servlet-based applications.

EJB technology is the J2EE's main component model for writing business logic. EJBs run inside specialized containers that provide infrastructure functionalities such as transaction management, security, concurrency control, and instance life cycle management, enabling developers to primarily focus on business functionality instead of low-level system issues. There have been three distinct bean types defined by the EJB specification, each serving different use cases: Session Beans that encapsulate business processes and client-facing services and are further class ified int o Stateless Session Beans, which maint ain no client-specific state between method invocations, Stateful Session Beans which maint ain client-specific state for the duration of a session, and Singleton Session Beans, which maint ain a

single instance per application and are useful when a shared state or coordinated operations are needed; Message-Driven Beans (MDBs), which offer message-oriented asynchronous processing by consuming messages from a JMS destination or message provider; and Entity Beans, which historically helped to provide object-relational mapping for database persistence but are now largely rendered obsolete by the Introduction of the Java Persistence API (JPA) since EJB 3.0. The development of EJB technology is a microcosm of the overall evolution of J2EE int o more developer-friendly programming models:[2] EJB 1.0 and 2.0 had long int erfaces, deployment descriptors, and lots of boilerplate code and were justly criticized for being complex and verbose; EJB 3.0 was a radical simplification thanks to annotations, dependency injection, and the Plain Old Java Object (POJO) programming model; this option drastically reduced development effort; new   er versions built on that with cleaner approaches and innovations like asynchronous method invocation, timer services, and better capability for transactions. EJBs inherently implement many of the foundational enterprise patterns: Component-Based Development uses a modular structure, Inversion of Control uses container-managed services, Dependency Injection uses resource acquisition, Facade Pattern for simplifying client access to complex subsystems, Business Delegate abstracts away remote implementation details. Although alternative frameworks such as Spring have captured much of the marketshare by providing equivalent functionality with reduced perceived overhead, EJBs are still a mainstay of many large enterprise applications, especially in cases where distributed transactions and complex security policies are involved or when int egrating with older legacy J2EE systems. An insight int o the component-based design concepts that are employed in a specific technology is useful—whether it be EJB or any future framework.

The Java Persistence API (JPA) is a specification that configures anObject Relational Mapping in the j2ee platform to provide a unified and object-oriented int erface to the relational data that can be managed as objects. Java Persistence API (JPA) was int roduced in EJB 3.0 to supersede the previous entity bean paradigm, which was criticized for its complexity and performance issues, and used a lightweight, Plain Old Java Object (POJO) setup leveraging proven Object Relational Mapping (ORM) frameworks like Hibernate. Essentially, JPA

reconciles the object-oriented world and the relational world using entities—plain old Java class es, annotated with @Entity, that correspond to persistent data structures. All these features are complemented with extra annotations to customize their mapping behavior: @Table for the database table or tables this entity is mapped to, @Id to identify primary key fields, @Column to configure the mapping of each single field, and relationship annotations (@OneToOne, @OneToMany, @ManyToOne, @ManyToMany) for the associations between entities. However, this doesn't cover the entire lifecycle of persistence. JPA empowers it with a richer set of features exposed via EntityManager instances that provide methods to persist, find, merge, and delete entities, while int ernally, it maint ains a persistence context that can track changes to an entity and propagate them to the underlying database. The specification defines a strict entity lifecycle: new /transient, managed, detached, removed – and transitions between them according to EntityManager operations and transaction boundaries. To retrieve data, JPA has several query methods: the Java Persistence Query Language (JPQL), a platform-independent, object-oriented query language that has the same building blocks as SQL but operates on entities rather than tables; the Criteria API, which is a type-safe, programmatic alternative to the String - based queries; and native SQL queries for accessing features that are only available in specific databases. It handles more sophisticated persistence issues such as inheritance mapping (with support for single table, joined table, and table-per-class strategies), composite keys, embedded objects, lazy loading of relationships, optimistic locking for concurrent access, and second-level caching for performance reasons. There are several JPA implementations available, including but not limited to Hibernate (the most popular), EclipseLink (JPA reference implementation), OpenJPA and others; however, they all wrap the standardized API and usually extend it with additional aspects/features. The JPA advancements over time and their new capabilities could be summarized as follows: JPA 2.0 brought the Criteria API, collection mappings, and validation int egrations; JPA 2.1 got stored procedures, fetching strategies and entity graphs, and attribute converters; JPA 2.2 int roduced support for some of the Java 8 features such as Stream API results, Date/Time types and repeatable annotations. However, since data persistence requirements are inherently a fundamental part of all

enterprise applications, JPA continues to be a cornerstone technology in the world of J2EE because it provides a very good blend of standardization and flexibility of database int egration for diverse scenarios.

The Java Message Service (JMS) resource adapter provides J2EE applications with standardized asynchronous messaging capabilities so that loosely-coupled communication is possible among distributed components across application boundaries. These messaging approaches provide additional benefits compared to synchronous communication approaches, including: temporal decoupling, where a sending application does not need to be online at the same time as the receiving application; load-leveling, where messages can be buffered for processing during variable workload periods; reliability, where the delivery of a message can be ensured and scaled across multiple consumers at ease using message-oriented middleware. JMS defines two main types of messaging models — and point -to-point (PTP) via queues where a message is sent to only one consumer instance, commonly used to perform a load balancing approach, and publish-subscribe (pub/sub) via topics where a message is sent to all active subscribers, well suited to event propagation or notifications distribution scenarios. The JMS API provides a uniform programming model across these patterns with a few principal int erfaces: ConnectionFactory and Connection for creating communication channels with the message provider, Session for creating messages and producers/consumers, MessageProducer for publishing messages to destinations, MessageConsumer for receiving messages from destinations, and various Message types (TextMessage, BytesMessage, MapMessage, StreamMessage, ObjectMessage) representing different payload formats. Messages are structured as not just payloads, but also headers (for standard routing and identification metadata) and properties (for application-specific attributes that aid in filtering and processing). JMS provides for synchronous consumption (the receiver instructs the provider to deliver a message), as well as for asynchronous consumption (messages trigger registered MessageListener callbacks), giving the application flexibility in what delivery model it chooses. Thus, J2EE's transaction model int egration allows messages to be part of distributed transactions, assuring that the messaging operations are consistent with other resources as databases.

Message-Driven Beans (MDBs) are a specific component model catering to message consumption, enabling developers to define the information processing without considering concurrency management, transaction management, and resource pooling, which are handled by the EJB container. Since its inception, JMS has been on an evolution path of simplification and int egration with other J2EE technologies: JMS 1.1 unified the separate point -to-point and publish-subscribe APIs; JMS 2.0 added a simplified API, delivery delay capabilities, and shared subscriptions for pub/sub load balancing across multiple consumers. While JMS standards have stood the test of time, as with many other legacy technologies, it is increasingly int egrated with (or replaced by) more modern messaging technologies, particularly in microservices or event-driven architectures context.

10.7: Security, Transactions, and Int egration in J2EE

Security is a key cross-cutting concern of the J2EE architecture and is handled through a broad architecture that cuts across all tiers and components of enterprise applications. The Model consists of different layers of security including authentication (verifying the identity of the user), authorization (access control to the resources), confidentiality (protection of data against disclosure), int egrity (data not altered during a transmission), and non-repudiation (a party cannot deny the authenticity of their signature). In bare terms, J2EE security implementations are normally conceived of as a combination of declarative where the constraint s are delineated via annotations or deploy descriptors with no touching of app code and programmatic where the security checks are embedded directly int o the business logic for int ricate access control. Authentication involves extracting credentials (for example through form-based login, HTTP Basic/Digest authentication, client certificates, single sign-on ticket, or int egrations to external systems such as LDAP, Kerberos, or SAML), validating the credentials based on user repositories, and issuing a security context to the authenticated session. User identities are grouped int o roles— logical groupings indicating application-specific functions or responsibilities—that access controls are defined against at a more role-based level to encourage maint ainability and scalability rather than granular definitions against individual user identities. Authorization constraint s can be imposed at various levels: web resources, using URL

patterns and HTTP methods; EJB methods based on callers' roles; application data that is filtered according to users' contexts; and even JMS destinations or web services that are offered only to authorized consumers. Container-managed services int egrate with the J2EE security model using the Java Authentication and Authorization Service (JAAS) to provide pluggable authentication modules, subject-based authorization, and delegation capabilities. For securing web services, specifications such as WS-Security provide the means for securing message-level protection, while for preventing the abuse of APIs, standards based on OAuth 2.0 and OpenID Connect are increasingly used in modern authentication scenarios. Transport-level security is usually built on TLS/SSL for secure communication, as the data should be encrypted when sent over the network; protecting data on the wire between tiers and to/from outside systems. Beyond these technical controls, robust J2EE security implementations must also mitigate concerns pertaining to secure configuration (removing default credentials and unnecessary services), input validation (to prevent injection attacks and cross-site scripting), session management (to guard against session fixation and session hijacking), auditing (to record security-relevant events for monitoring and compliance purposes), and secure exception handling (to avoid information leaks in error messages). J2EE security has evolved alongside new threats and new deployment patterns: Java EE 6 brought programmatic login and int erceptor-based security; Java EE 7 added expression-based access control support; Java EE 8 int roduced a new Security API (JSR 375) that made it easier to configure identity stores and HTTP-authentication mechanisms; and Jakarta EE has continued to build upon these abilities to support cloud-native and microservices environments.

Transaction management is one of the several most significant infrastructural services provided by J2EE that offers the ability to help ensure data consistency and int egrity across many operations and resources. ACID — Atomicity (all or nothing); Consistency (A transaction should maint ain the data in a valid state before and after the execution); Isolation (As an impact of the operation will not alter the rest of the transactions); and Durability (the committed changes persist during failure cases). J2EE provides two basic transaction management styles: container-managed transactions (CMT), where the

application server automatically manages transaction demarcation based on declarative configurations, and bean-managed transactions (BMT), where application code explicitly controls transaction boundaries. In the case of container-managed transactions, the developer indicates transaction attributes that describe how components participate in transactions: Required creates a new transaction or join an existing transaction if one exists; RequiresNew always creates a new transaction; Mandatory requires an existing transaction; NotSupported suspends any current transaction; Supports joins an existing transaction but does not require one; and Never prohibits being run within a transaction context. You can set these attributes through annotations (@TransactionAttribute) or deployment descriptors, giving you fine-grained control (without peppering your rigid business code with transaction details). One especially powerful feature of J2EE is its support for distributed transactions (also known as global or XA transactions) across multiple heterogeneous resources including databases, message queues, and legacy systems. As in other transactional systems, the ability to coordinate commits across resources is provided by the transaction manager and the two-phase commit (2PC) protocol, as users join the transaction using the Java Transaction API (JTA) to ensure atomicity across participating resources. Resource int egration is done using J2EE resource adapters, which implement the XA int erface, providing the ability for transaction manager to enlist such resources in distributed transactions. Transaction management generally int eracts with other container services. Although the J2EE transaction model is a great fit for consistency in traditional applications, it struggles with distributed cloud architectures where we can see several issues, like the performance impact due to distributed transactions, and the fact that ACID guarantees are not useful with long-running workflows. As a result, modern J2EE applications typically layer optional eventual consistency patterns, compensating transactions, or saga patterns on top of ACID transactions for certain distributed cases, even though the transaction infrastructure platform is very much basic for those core business operations in which we can't compromise on data int egrity.

Tight int egrations with Enterprise Information Systems (EIS), such as ERP systems, mainframe applications, database systems, and other legacy infrastructure is done through J2EE's standardized approach,

provided here by the Java Connector Architecture (JCA). Before JCA, the int egration did not usually use any standards, and relied heavily on custom-built, point -to-point connectors that led to maint enance and duplication nightmares in a multi-project environment. JCA solves these problems by defining a common architecture of resource adapters, which are specialized components that serve as a bridge between J2EE applications and resource managers (such as database connection pools, EISes, or messaging systems); these components make use of the services provided by the container (transaction management, security, connection pooling). This architecture consists of three main contracts: the Connection Management contract, which defines the central pooling, lifecycle management, and allocation optimization models when connections to database servers are made; the Transaction Management contract, which allows resource adapters to participate in container-managed transactions by coordinating both local and XA transactions; and the Security contract, which provides secure access to external systems by mapping credentials, delegating principal, and propagating the security context. JCA resource adapters would generally have a standard Common Client Int erface (CCI) for application code want to talk to the EIS, and adapter-specific int erfaces that are specific to the external systems. This enables application servers to cater to different int egration scenarios and ensure similar management approaches across various types of EIS connections. In addition to basic connectivity, JCA also supports different patterns of int eraction: synchronous request-reply for operations that require an immediate response, local transactions for simple consistency requirements, distributed transactions for operations that span multiple resources and record-based int erfaces for structured data exchange. The spec has matured to meet the increasing int egration challenges: JCA 1.5 had work management for incoming communication, creating message endpoint s that consume events from thirdparty sources; JCA 1.6 included support for annotations, pluggable work contexts, and better lifecycle management features; and JCA 1.7 enhanced security and connection validation capabilities. Although JCA is a thorough int egration solution, other paths are open in the J2EE world: Web Services (JAX-WS, JAX-RS) is a common for service-oriented int egration, JMS is for message-oriented middleware, JDBC is a low-level access to databases, and Java API for XML Processing (JAXP) is for XML-

oriented data int erchanging. With API-based int egration, lightweight REST services, and cloud-native connectivity becoming the order of the day, JCA is not given the same prominence in new application development as it might have been in the past. Nevertheless, JCA is still critical for int egration with legacy systems in enterprises where there are no alternatives available. Learning the concepts behind JCA is needful to know about enterprise int egration patterns and different challenges to address this int egration, no matter what particular technology used to accomplish this goal.

J2EE Web Services technologies allow distributed applications to communicate over the platform and organizational boundaries in an int eroperable and cross heterogeneous environment. The platform is based on two styles of web service generation: SOAP with XSD/WSDL documents and REST with standard HTTP verbs and semantics. Center of SOAP based development, the Java API for XML Web Services (JAX-WS) serves a powerful API which uses annotations and auto-generated artifacts to make service implementation much easier. It is possible for developers to expose services simply by annotating a class with @WebService and methods with @WebMethod; the container will generate the required WSDL, XML Schema definitions, and marshalling code. JAX-WS also well adapts to both approaches to handling WSDL files: top-down (starting from existing WSDL documents) and bottom-up (where WSDL will be generated from the Java class es). Example 1: Java Architecture for XML Binding (JAXB) JAXB handles complex data types and maps them to and from Java class es automatically. JAXB does the marshalling and unmarshalling of Java object to XML and back to Java automatically. Widespread in enterprise scenarios, JAX-WS is extended with WS-Security for message-level security, WS-ReliableMessaging for guaranteed delivery, WS-Addressing for asynchronous communication and WS-Policy for declarative configuration. For REST-based services, there is the Java API for RESTful Web Services (JAX-RS) which is a lightweight specification that focuses on dealing with resources and HTTP key concepts in an annotation-based programming model. Resource class es are annotated with @Path to specify URI patterns and methods are further annotated with @GET, @POST, @PUT, or @DELETE to specify which HTTP operation they143565iws seventeen220 query for. Content negotiation

occurs via the @Produces and @Consumes annotations, where you indicate the acceptable media types, while parameters get bound as per annotations such as @PathParam, @QueryParam and @FormParam. JAX-RS uses serialization and deserialization for Java objects and many other representations such as JSON, XML, text, and so on based on content negotiation. In addition to these key specifications, the J2EE web services ecosystem provides supporting technologies such as JSON Processing (JSON-P) and JSON Binding (JSON-B) for working with structured data, WebSocket API for bidirectional communication, and Concurrency Utilities for asynchronous processing. The trajectory of J2EE web service evolutions mirrors broader industry directions: in its early days, J2EE support was focused on SOAP and WS-* specifications for enterprise int egration, with Java EE 6 adding robust RESTful support in JAX-RS 1.1, and Java EE 7 improving both paradigms with client APIs and more format options, but Java EE 8 and Jakarta EE have increasingly favored lightweight, cloud-friendly approaches prioritizing REST, JSON and reactive programming models. And because APIs will become the very

Check Your Progress

1. **What is the main difference between Java SE and J2EE?**

    ……………………………………………………………………
    ……………………………………………………………………
    ……………………………………………………………………
    ……………………………………………………………………

2. **What is a "container" in the J2EE architecture?**

    ……………………………………………………………………
    ……………………………………………………………………
    ……………………………………………………………………
    ……………………………………………………………………

10.8: Summary

This unit int roduced J2EE (now Jakarta EE) as an enterprise platform for building large-scale, server-side applications. We learned that J2EE is a specification that extends Java SE with a set of powerful APIs. Its architecture is typically divided int o multiple tiers: the **Client Tier**

(like a browser), the **Web Tier** (handling HTTP requests with Servlets/JSP), the **Business Tier** (running complex business logic with EJBs), and the **Enterprise Information System (EIS) Tier** (connecting to databases or legacy systems). The magic of J2EE lies in its **containers** (like the Web Container and EJB Container), which are runtime environments that provide core services like life cycle management, security, and transaction control, allowing developers to focus purely on business logic.

10.9: Exercises

Multiple Choice Questions

1. What is J2EE primarily used for?

    a) Creating desktop GUI applications.

    b) Building large-scale, server-side enterprise applications.

    c) Writing low-level operating system drivers.

    d) Developing mobile apps for Android.

    Answer: (b) Building large-scale, server-side enterprise applications.

2. In the J2EE multi-tier architecture, which tier is responsible for handling HTTP requests and responses?

    a) Client Tier

    b) Business Tier

    c) Web Tier

    d) EIS Tier

    Answer: (c) Web Tier

3. The J2EE component that provides runtime services like security, transaction management, and life cycle management is called a:

    a) Container

    b) Compiler

    c) JVM

    d) Package

    Answer: (a) Container

4. Which of the following is NOT typically considered a core J2EE technology?

    a) JavaFX

    b) Servlets

c) JavaServer Pages (JSP)

d) Enterprise JavaBeans (EJB)

Answer: (a) JavaFX

5. In a multi-tier model, the database is part of the:

a) Client Tier

b) Business Tier

c) Web Tier

d) Enterprise Information System (EIS) Tier

Answer: (d) Enterprise Information System (EIS) Tier

Descriptive Questions:

1. Draw a diagram of the standard J2EE four-tier architecture. Label each tier and give one example of a technology that runs in it.

2. What is a J2EE "container"? Explain the difference between a Web Container and an EJB Container.

3. Why is a multi-tier architecture beneficial for enterprise applications? Discuss at least two advantages (e.g., scalability, separation of concerns).

4. What is the relationship between Java SE and J2EE? Can you have J2EE without Java SE?

5. What is an "Application Server" (like GlassFish, WildFly, or WebSphere) and what is its role in the J2EE ecosystem?

10.10: References and Suggested Readings

- Eclipse Foundation. (n.d.). *Jakarta EE official documentation*. Retrieved from https://jakarta.ee (Official specifications and tutorials)

- Basham, B., Sierra, K., & Bates, B. (2008). *Head first servlets & JSP* (2nd ed.). O'Reilly Media.

- Keogh, J. (2002). *J2EE: The complete reference*. McGraw-Hill.

- Oracle. (n.d.). *Java EE tutorials* (Archived). Oracle Documentation.

Block 3: Servlet Technology

Unit 11: Java Servlet

Structure

11.1: Introduction

When you submit a form on a website, a program on the server needs to receive that data, process it, and send a response back (often as a new HTML page). A **Java Servlet** is the core J2EE technology that does exactly this. A Servlet is a Java program that runs inside a **Web Container** (part of a web server or application server). It is the engine that handles HTTP requests and generates dynamic responses. Unlike older technologies like CGI, Servlets are efficient because they are loaded once and can handle many requests simultaneously. This unit int roduces the Servlet API, showing how to create a simple servlet that can read data from an HTTP request and write an HTML page back as a response.

11.2: Learning Outcomes

After completing this unit, you will be able to:

1. Define "Servlet" and its role in a web application.

2. Understand the request/response model of web communication (HTTP GET/POST).

3. Create a basic servlet by extending the HttpServlet class .

4. Implement the doGet() and doPost() methods to handle different request types.

5. Read data from an HttpServletRequest object.

6. Write a response (e.g., HTML) back to the client using the HttpServletResponse object.

7. Understand the purpose of the web.xml (Deployment Descriptor) file.

11.3: Java Servlet: Basic Servlet Structure
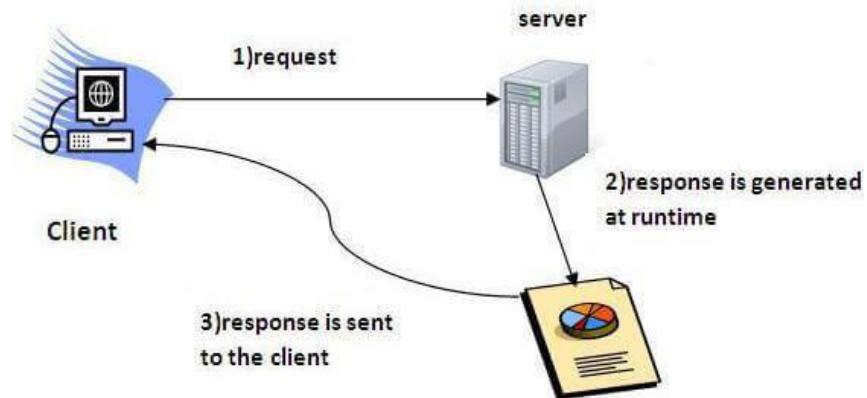
Figure 11.1: Servlet

Java Servlets are one of the key technologies of Java web development technology, they are the basis of server-side programming in Java, which with the emergence of many clients does not lose its popularity. Servlets are basically Java class es made with the purpose of following the given specification of Java ServletAPI to handle request and generate response normally inside a/your web application framework. Servlet technology dates from the late 1990s as one of Java's first enterprise offerings, responding to the shortcomings of CGI (Common Gateway Int erface) programming by providing higher performance, platform independence, and to easily take advantage of the Java ecosystem. Servlets run inside servlet containers (or web containers) that provide the runtime environment and lifecycle management. With the container-based architecture, infrastructure management and application logic are separated, giving developers the freedom to focus on business functionality rather than lower-level protocols and communication mechanisms. Where CGI-based programs create a new process for each request, servlets run inside the JVM, which provides sophisticated support for multi-threading. This underlying architectural difference allows servlets to offer much better performance and resource usage than older web programming models. Even though more abstracted frameworks such as JavaServer Pages (JSP),

JavaServer Faces (JSF), and various other MVC implementations followed in its wake, servlets are the real based technology behind Java web applications. For any Java developer who is working on web application, understanding servlets is a prerequisite, since all high level frameworks are finally backed by the servlet technology, behind the scenes, Servlet technology is the core of all request-response mechanism. The servlet spec has come a long way since it was int roduced, and in each version, new   features have been added but are still backward compatible. New   er servlet implementations offered support for annotations, async processing, non-blocking I/O and other improvements which have helped keep this technology useful in modern web development contexts.

11.4: Basic Structure and Core Components of Java Servlets

A servlet is a simple Java class that has to extend appropriate servlet class (javax.servlet.Servlet) and then implement specific methods that handle the request from the client. All servlets must implement the javax. servlet. Servlet int erface - This int erface defines all the necessary methods needed for the servlet lifecycle management and the request processing. However, rather than implementing the Servlet int erface directly, most developers extend the GenericServlet or HttpServlet abstract class es, which provide partial implementations of the int erface. In particular, the HttpServlet class is important because it is used to handle HTTP-specific request-response int eractions using methods like doGet(), doPost(), doPut(), doDelete(), etc., corresponding to the HTTP methods. The following steps summarize the typical structure of a servlet implementation: package declarations, imports, non-required annotations, class   declaration extending HttpServlet, non-required constructors, must-have lifecycle methods (init, destroy), and must-have request handler methods. Servlet structure: A servlet contains a variety of structural components such as deployment descriptors (specifically specification web. xml (or using annotations in modern implementations), servlet mappings to associate URL patterns with servlet instances, initialization parameters that configure how servlets behave, and context parameters that are applicable across the entire web application. Request handling methods are at the heart of servlet functionality and accept

HttpServletRequest and HttpServletResponse objects as their parameters – these objects are the primary conduits for int eraction with clients. The request object contains all the information the client sent to the server, such as parameters, headers, cookies, and session data, and the response object has methods to set data to be sent to the client, set response headers, set cookies, and control the status of the response. Servlets are inherently multithreaded, meaning that it is important to consider how to handle multithreading in the design of a servlet; the servlet container instantiates a single instance of a servlet and then handles multiple requests to the servlet by invoking them on multiple threads, which means that thread safety is paramount. Servlet error handling uses Java's exception mechanism, but it has special rules for catching and reporting checked and runtime exceptions. The servlet architecture also includes request filtering capabilities through the Filter API, allowing pre-processing and post-processing operations to be applied across servlets, and the use of listeners to handle various events occurring in the application or user session. Grasping these structural char acteristics establishes the groundwork required for successful servlet programming, allowing developers to design solid, maint ainable web applications that effectively utilize the features of the Java Servlet API.

CGI (Common Gateway Int erface)

CGI technology enables the web server to call an external program and pass HTTP request information to the external program to process the request. For each request, it starts a new process.
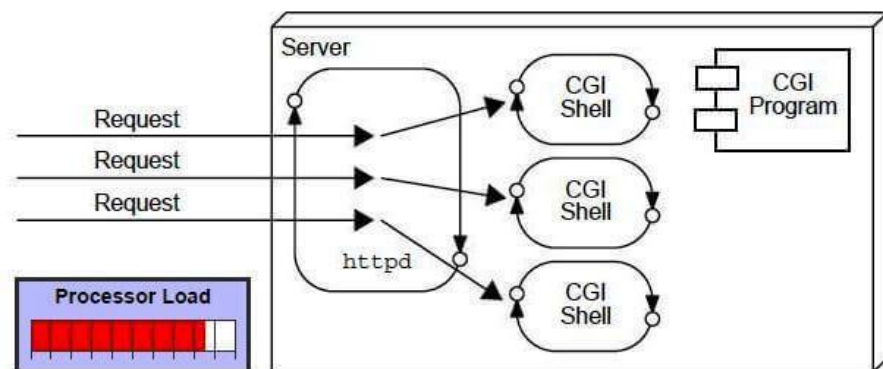
Figure 11.2: CGI

Disadvantages of CGI

There are many problems in CGI technology:

1. If the number of clients increases, it takes more time for sending the response.
2. For each request, it starts a process, and the web server is limited to start processes.
3. It uses platform dependent language e.g. C, C++, perl.

Advantages of Servlet

There are many advantages of Servlet over CGI. The web container creates threads for handling the multiple requests to the Servlet. Threads have many benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low. The advantages of Servlet are as follows:
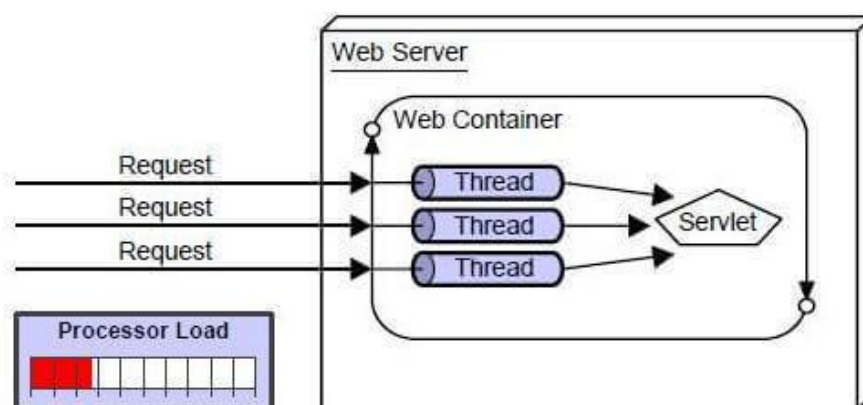
Figure 11.3: Servlet

1. **Better performance:** because it creates a thread for each request, not process.
2. **Portability:** because it uses Java language.
3. **Robust:** JVM manages Servlets, so we don't need to worry about the memory leak, garbage collection, etc.
4. **Secure:** because it uses java language.

Web Terminology

Servlet Terminology	Description

Website: static vs dynamic	It is a collection of related web pages that may contain text, images, audio and video.
HTTP	It is the data communication protocol used to establish communication between client and server.
HTTP Requests	It is the request send by the computer to a web server that contains all sorts of potentially int eresting information.
Get vs Post	It gives the difference between GET and POST request.
Container	It is used in java for dynamically generating the web pages on the server side.
Server: Web vs Application	It is used to manage the network resources and for running the program or software that provides services.
Content Type	It is HTTP header that provides the description about what are you sending to the browser.

Steps to create a servlet example

There are given 6 steps to create a servlet example. These steps are required for all the servers.

The servlet example can be created by three ways:

1. By implementing Servlet int erface,
2. By inheriting GenericServlet class , (or)
3. By inheriting HttpServlet class

The mostly used approach is by extending HttpServlet because it provides http request specific method such as doGet(), doPost(), doHead() etc.

Here, we are going to use apache tomcat server in this example.

The steps are as follows:

1. Create a directory structure
2. Create a Servlet
3. Compile the Servlet
4. Create a deployment descriptor
5. Start the server and deploy the project
6. Access the servlet

1)Create a directory structures

The directory structure defines that where to put the different types of files so that web container may get the information and respond to the client.

The Sun Microsystem defines a unique standard to be followed by all the server vendors. Let's see the directory structure that must be followed to create the servlet.
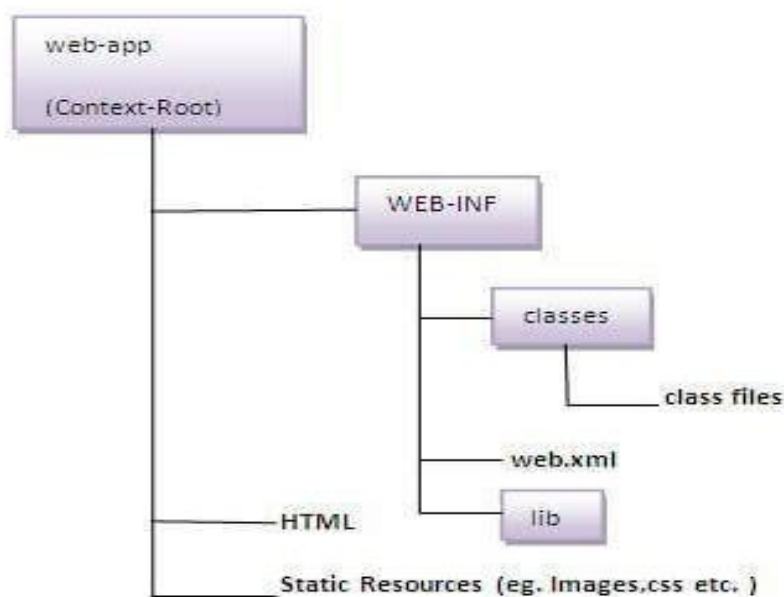
Figure 11.4: Directory Structure

As you can see that the servlet class file must be in the class es folder. The web.xml file must be under the WEB-INF folder.

2)Create a Servlet

There are three ways to create the servlet.

1. By implementing the Servlet int erface
2. By inheriting the GenericServlet class
3. By inheriting the HttpServlet class

The HttpServlet class is widely used to create the servlet because it provides methods to handle http requests such as doGet(), doPost, doHead() etc.

In this example we are going to create a servlet that extends the HttpServlet class . In this example, we are inheriting the HttpServlet class and providing the implementation of the doGet() method. Notice that get request is the default request.

DemoServlet.java

1. import javax.servlet.http.*;
2. import javax.servlet.*;
3. import java.io.*;
4. public   class DemoServlet extends HttpServlet{
5. public   void   doGet(HttpServletRequest req,HttpServletResponse res)
6. throws ServletException,IOException
7. {
8. res.setContentType("text/html");*//setting the content type*
9. Print
   Writer pw=res.getWriter();*//get the stream to write the data*
10.
11. *//writing html in the stream*
12. pw.print ln("<html><body>");
13. pw.print ln("Welcome to servlet");
14. pw.print ln("</body></html>");
15.
16. pw.close();*//closing the stream*
17. }}

3)Compile the servlet

For compiling the Servlet, jar file is required to be loaded. Different Servers provide different jar files:

Jar file	Server
1) servlet-api.jar	Apache Tomcat
2) weblogic.jar	Weblogic
3) javaee.jar	Glassfish
4) javaee.jar	JBoss

Two ways to load the jar file
1. set class path
2. paste the jar file in JRE/lib/ext folder

Put the java file in any folder. After compiling the java file, paste the class file of servlet in WEB-INF/class es directory.

4)Create the deployment descriptor (web.xml file)

The deployment descriptor is an xml file, from which Web Container gets the information about the servet to be invoked.

The web container uses the Parser to get the information from the web.xml file. There are many xml parsers such as SAX, DOM and Pull. There are many elements in the web.xml file. Here is given some necessary elements to run the simple servlet program.

web.xml file
1. <web-app>
2.
3. <servlet>
4. <servlet-name>sonoojaiswal</servlet-name>
5. <servlet-class >DemoServlet</servlet-class >
6. </servlet>
7.
8. <servlet-mapping>
9. <servlet-name>sonoojaiswal</servlet-name>
10. <url-pattern>/welcome</url-pattern>
11. </servlet-mapping>
12.
13. </web-app>

Description of the elements of web.xml file

There are too many elements in the web.xml file. Here is the illustration of some elements that is used in the above web.xml file. The elements are as follows:

<web-app> represents the whole application.

<servlet> is sub element of <web-app> and represents the servlet.

<servlet-name> is sub element of <servlet> represents the name of the servlet.

<servlet-class > is sub element of <servlet> represents the class of the servlet.

<servlet-mapping> is sub element of <web-app>. It is used to map the servlet.

<url-pattern> is sub element of <servlet-mapping>. This pattern is used at client side to invoke the servlet.

5)Start the Server and deploy the project

To start Apache Tomcat server, double click on the startup.bat file under apache-tomcat/bin directory.

One Time Configuration for Apache Tomcat Server

You need to perform 2 tasks:

1. set JAVA_HOME or JRE_HOME in environment variable (It is required to start server).
2. Change the port number of tomcat (optional). It is required if another server is running on same port (8080).

1) How to set JAVA_HOME in environment variable?

To start Apache Tomcat server JAVA_HOME and JRE_HOME must be set in Environment variables.

Go to My Computer properties -> Click on advanced tab then environment variables -> Click on the new   tab of user variable -> Write JAVA_HOME in variable name and paste the path of jdk folder in variable value -> ok -> ok -> ok.

Go to My Computer properties:

Click on advanced system settings tab then environment variables:
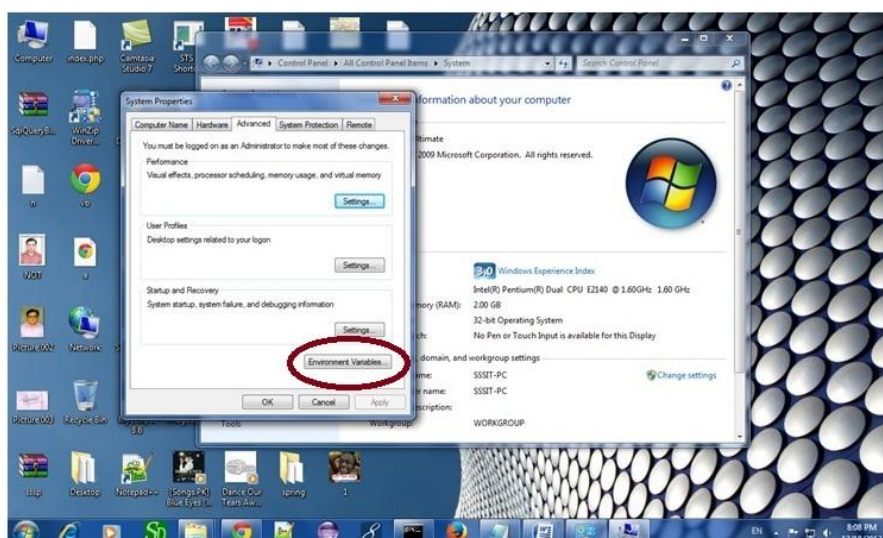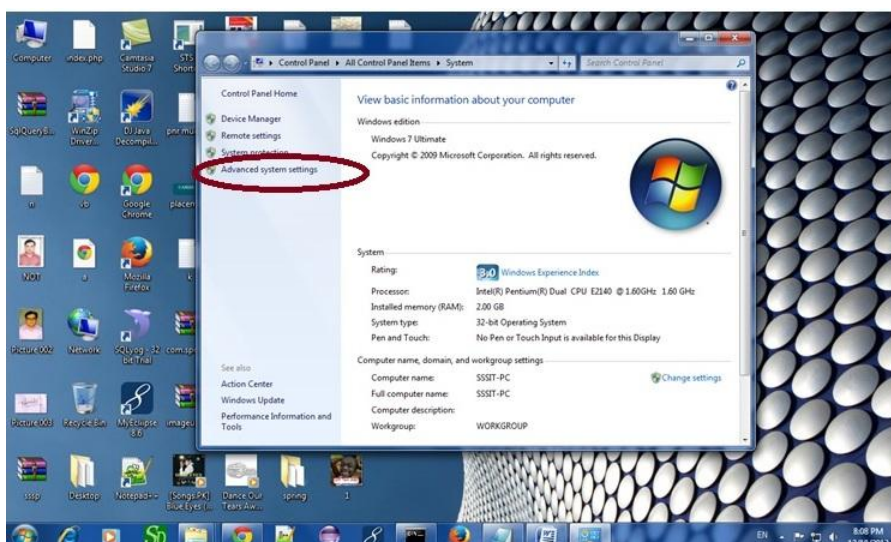
Click on the new   tab of user variable or system variable:
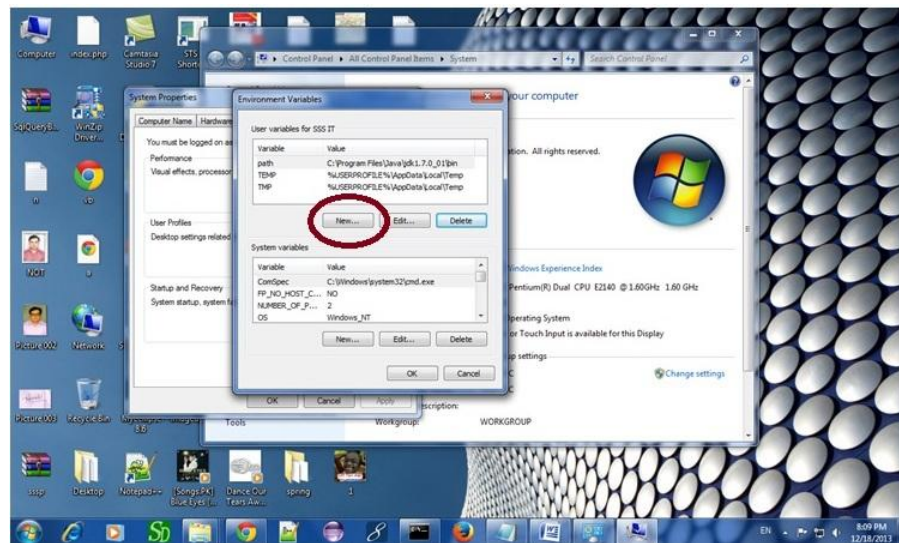
Write JAVA_HOME in variable name and paste the path of jdk folder in variable value:
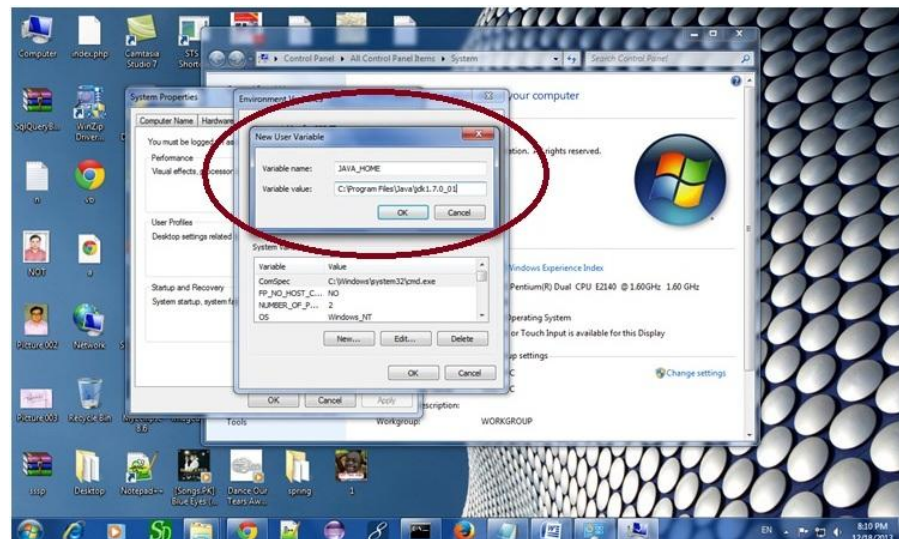
There must not be semicolon (;) at the end of the path.

After setting the JAVA_HOME double click on the startup.bat file in apache tomcat/bin.

Note: There are two types of tomcat available:

1. Apache tomcat that needs to extract only (no need to install)
2. Apache tomcat that needs to install

It is the example of apache tomcat that needs to extract only.
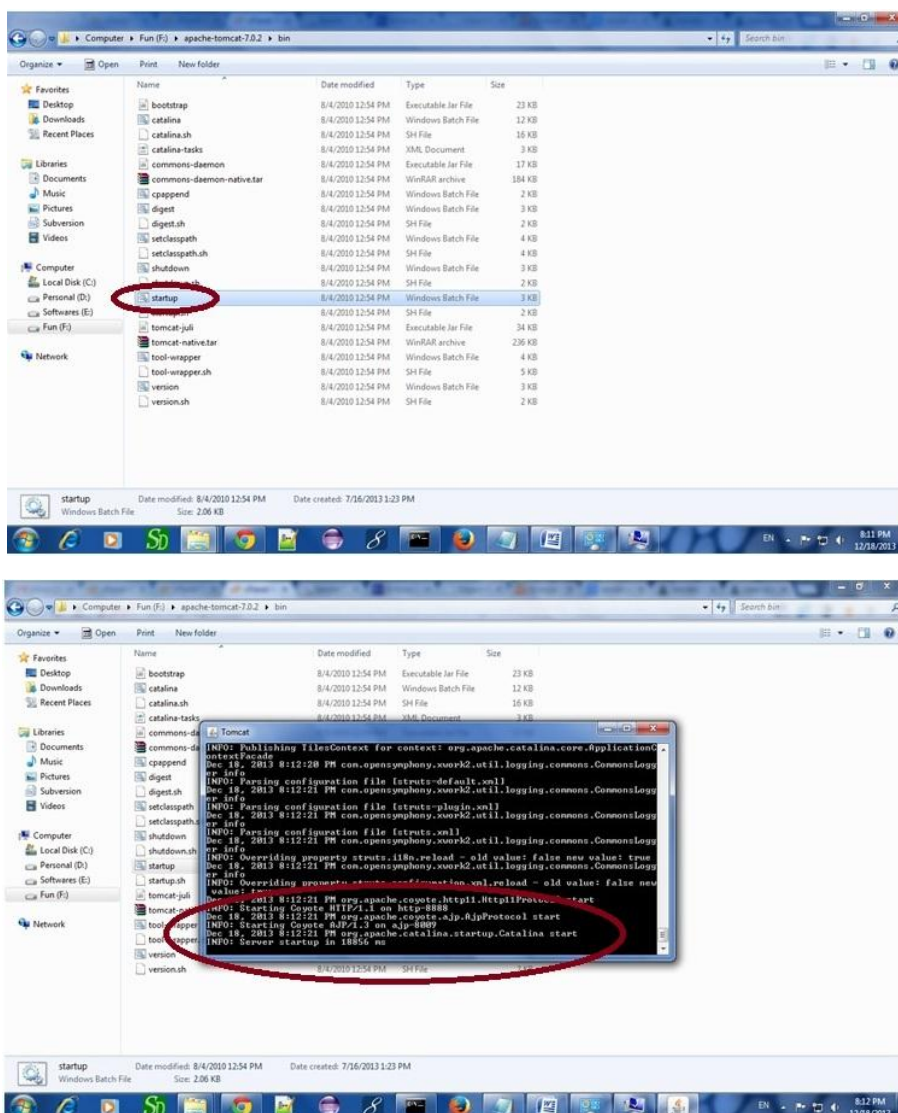
Now server is started successfully.

2) How to change port number of apache tomcat

Changing the port number is required if there is another server running on the same system with same port number. Suppose you have installed oracle, you need to change the port number of apache tomcat because both have the default port number 8080.

Open server.xml file in notepad. It is located inside the apache-tomcat/conf directory . Change the Connector port = 8080 and replace 8080 by any four digit number instead of 8080. Let us replace it by 9999 and save this file.

5) How to deploy the servlet project

Copy the project and paste it in the webapps folder under apache tomcat.
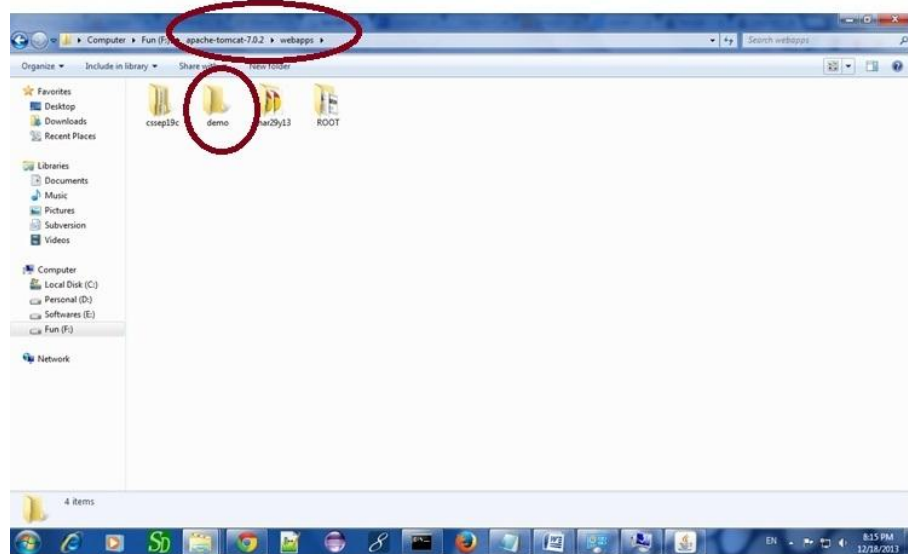
But there are several ways to deploy the project. They are as follows:

- By copying the context(project) folder int o the webapps directory
- By copying the war folder int o the webapps directory
- By selecting the folder path from the server
- By selecting the war file from the server

Here, we are using the first approach.

You can also create war file, and paste it inside the webapps directory. To do so, you need to use jar tool to create the war file. Go inside the project directory (before the WEB-INF), then write:

projectfolder> jar cvf myproject.war *

Creating war file has an advantage that moving the project from one location to another takes less time.

6) How to access the servlet

Open broser and write-

http://hostname:portno/contextroot/urlpatternofservlet.

For example:

1. http://localhost:9999/demo/welcome

Welcome to servlet
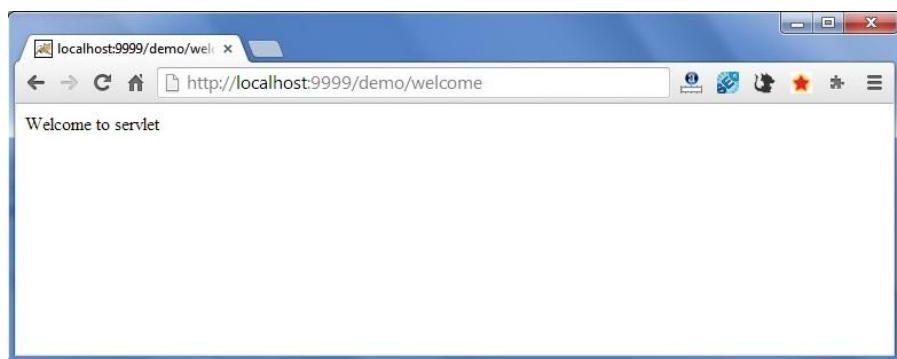
Check Your Progress

1. **What is a Java Servlet?**

   ……………………………………………………………
   ……………………………………………………………
   ……………………………………………………………
   ……………………………………………………………

   2. What is the difference between an HTTP GET request and a POST request?

   ……………………………………………………………
   ……………………………………………………………
   ……………………………………………………………
   ……………………………………………………………

11.5: Summary

This unit int roduced the Java Servlet, the fundamental component for server-side web programming in Java. We learned that a servlet is a Java object that runs in a web container and is responsible for handling HTTP requests. By extending HttpServlet, we can override the doGet() and doPost() methods to process data from a web form or URL. The HttpServletRequest object gives us access to all incoming data (like parameters and headers), while the HttpServletResponse object allows us to send a dynamically generated response (like HTML) back to the user's browser. Finally, we learned that servlets are mapped to specific URL patterns using either annotations or the web.xml deployment descriptor.

11.6: Exercise

Multiple Choice Questions

1. Which class must a servlet typically extend to handle web requests?
   a) GenericServlet
   b) Servlet
   c) HttpServlet
   d) JspServlet

   Answer: (c) HttpServlet

2. To handle an HTTP GET request, which method should you override?
   a) doGet()
   b) doPost()
   c) service()
   d) init()

   Answer: (a) doGet()

3. Which object is used to read incoming data, like form parameters, in a servlet?
   a) HttpServletResponse
   b) ServletConfig
   c) ServletContext
   d) HttpServletRequest

   Answer: (d) HttpServletRequest

4. Which object is used to send an HTML page back to the browser?
   a) HttpServletResponse
   b) HttpServletRequest
   c) ServletOutputStream
   d) Print Writer (obtained from the response)

   Answer: (a) HttpServletResponse

5. The file used to map a servlet to a URL pattern in older applications is:
   a) config.xml
   b) web.xml (Deployment Descriptor)
   c) servlet.properties
   d) httpd.conf

   Answer: (b) web.xml (Deployment Descriptor)

Descriptive Questions

1. What is the difference between a GET and a POST request? When should you use each?

2. Explain the role of the Web Container in relation to a servlet. What services does it provide?

3. Write a simple "Hello World" servlet that responds with the HTML "Hello World".

4. How do you read form data in a servlet? Write a code snippet that retrieves a parameter named "username" from an HTTP POST request.

5. What is the web.xml file (Deployment Descriptor)? What is its purpose? (Note: also mention that modern apps can use @WebServlet annotation).

11.7: References and Suggested Readings

- **Head First Servlets & JSP** by Bryan Basham, Kathy Sierra & Bert Bates (The definitive guide).

- **Java: The Complete Reference** by Herbert Schildt (Chapters on Servlets).

- **Jakarta EE (J2EE) Official Documentation:** (Servlet specifications).

- **Oracle's Java EE Tutorials:** (Archived tutorials on Servlets).

Block 3: Servlet Technology

Unit 12: Servlet Life Cycle

Structure

12.1: Introduction

Servlets are server-side Java programs that handle client requests and generate dynamic responses, forming the backbone of Java-based web applications. A servlet operates within a Servlet container (like Apache Tomcat) and follows a well-defined life cycle managed by the container. The servlet life cycle includes stages such as loading, initialization, request handling, and destruction.

Servlets communicate using the HTTP protocol, handling requests (HttpServletRequest) and responses (HttpServletResponse) through methods like doGet() and doPost(). They play a key role in session management, allowing web applications to maint ain user state across multiple int eractions using mechanisms like cookies, URL rewriting, or HttpSession.

Security is a critical aspect of servlet-based applications, involving authentication, authorization, and data confidentiality. Servlets are also part of the larger Java EE (now Jakarta EE) ecosystem, int egrating seamlessly with technologies such as JSP, JDBC, and EJB for enterprise-level solutions. Understanding how servlets handle form

data, manage sessions, and ensure secure communication is essential for developing scalable and secure Java web applications.

12.2: Learning Outcomes

After completing this unit, learners will be able to:

1. Describe the life cycle of a servlet and explain each stage in servlet execution.
2. Understand HTTP request and response processing in servlets.
3. Implement session management and maint ain user state across multiple requests.
4. Explain servlet security mechanisms, including authentication and authorization.
5. Int egrate servlets within the Java EE/Jakarta EE ecosystem for enterprise applications.
6. Demonstrate how to read and process form data using servlet request parameters.
7. Apply session tracking techniques such as cookies, URL rewriting, and HttpSession.

12.3: Servlet Life Cycle

This servlet life cycle governs how servlets are created, initialised, subserviced request and finally destroyed within the container environment. Servlet life cycle is a step-by-step process of such states followed by transitions which is handled by only the servlet container which calls certain methods at specific time on the servlet. From when web container either loads the servlet class (when web app starts) or
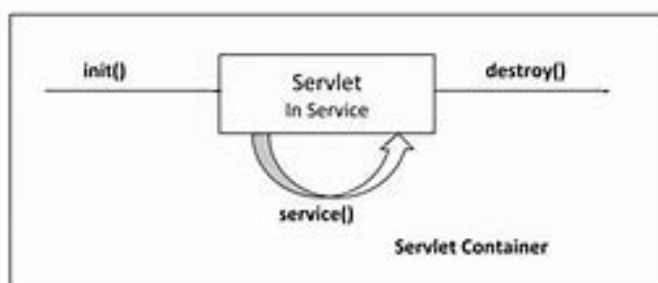
figure 12.1: Servlet Life Cycle

when first request comes (dependent on load-on-startup). Once your class has been loaded, a container shall instantiate one and only one instance of a servlet to your no-argument constructor, making it a

singleton with respect to your application. After that comes the initialization phase, during which the container calls the servlet's init(ServletConfig config) method and passes it a ServletConfig object that allows access to initialization parameters and the ServletContext. This initialization action is critical for execution of resource expensive tasks such as examples are database connection establishment, configuration file reading, or other setup processes. The init() method completes before the servlet can attempt to handle client requests. After initialization, the servlet goes int o the service phase: it lives on and responds to client requests until the container removes it. In this state, each request from a client causes the container to call the servlet's service() method (or, for HTTP servlets, the appropriate HTTP method handler such as doGet() or doPost()) on possibly multiple threads. Since servlets are singletons, instance variables of the servlet might be accessed by multiple request-processing threads concurrently, thus making thread-safety an important consideration for servlet implementations. When the Times of retting a servlet out of servile — when the application shuts down, or redeploys the server — whether the servlet is a phase of destruction by its destroy() method. This makes it possible to free resources, close connections, and perform other cleanup operations. The servlet lifecycle ends after destruction, when the instance switches to being eligible for garbage collection. This lifecycle is managed by the servlet container, the part of the web server that handles the servlet's functionality, which serves the dual role as manager of the servlet's execution environment by encapsulating communication protocols, implementing thread management, enforcing security policies, and providing pooling of servlet resources, freeing the servlet developers to concentrate on business logic rather than the int ricacies of infrastructure. The container makes sure the contract defined by the Servlet API is followed by instantiation of request and response objects, handling session tracking, enforcing security constraint s, and enabling access to shared resources by way of the ServletContext. This container-servlet synergy illustrates the class ic "inversion of control" paradigm (the container calls servlet methods, not the other wayaround); which leads to a simplified, generic, and standardized component model that's perfectly tailored for enterprise-level applications.

12.4: Request Processing and HTTP Handling in Servlets

This is the basic functionality of servlet, taking client requests and generating response for those requests. A web application reacts to an HTTP request it receives from a client by dispatching that request to the servlet container, which then forwards that to the servlet for processing, determining the proper servlet that can fulfil the request by using URL mapping configurations defined in a deployment descriptor or by annotation-based configurations. Once the container has determined which HttpServlet will service the request, it creates the HttpServletRequest and HttpServletResponse objects which encapsulate the data of the client's request and the means to formulate a response to the request, respectively. The service method of the servlet is where the request is passed to the appropriate HTTP method handler (such as GET, POST, PUT, or DELETE) for the HTTP method being made. The HttpServletRequest int erface provides you with full access to all parts of the incoming request

12.5: Session Management and State Persistence

Sending cryptographic hashes of credentials and Client Certificate Authentication (using X.509 certificates) as well as programmatic authentication through the HttpServletRequest. Servlet 3.0 The login() method that we can use. After the user has been authenticated, identity information is made available to servlets using methods such as getUserPrincipal(), getRemoteUser(), and isUserInRole(), which can support fine-grained, role-based access control within application code. Transport Layer Security (TLS/SSL) can provide confidentiality and int egrity protection for servlet communications; configurability is done through a element in security constraint s. In addition to declarative security, servlets support programmatic security via the previous HttpServletRequest methods and the new er SecurityContext API. For example, cross-site scripting (XSS) protection mechanisms include output encoding utilities and the HttpOnly and Secure cookie attributes, while cross-site request forgery (CSRF) defenses may rely on synchronizer tokens that servlets can generate and validate. Servlet A servlet is a purely Java class that extends the capabilities of a server, like a web server. Java servlets solve this problem through a series of complementary session management and state persistence mechanisms. The core mechanism

for session tracking is the HttpSession int erface, which acts as a server-side container that stores and allows retrieval of information specific to a user across multiple requests. The first time that a client accesses the application, the servlet container sends the client a unique session identifier, using either cookies or URL rewriting, and binds an HttpSession object (using the identifier) to the client. On subsequent requests, the container retrieves the session identifier from the client, finds a corresponding HttpSession, and makes it accessible to servlets via the getSession() method of HttpServletRequest. A session object acts as a key-value store, allowing servlets to insert attribute objects with setAttribute(String name, Object value) method at the same time repopulating the requests from the same client using the method getAttribute(String name), thus preserving state across requests from the same client. The container automatically manages the session lifecycle, creating sessions on-demand, maint aining session activity and invalidating them after a specified timeout period or by the application when the session invalidate() method is called. The servlet specification describes some session tracking mechanisms, including (the default and most common way) cookies, URL rewriting (attaching the session-id at the end of the URL when cookies are disabled), secure sockets layer (SSL) session information, and (outdated) hidden form fields. In addition to session management, servlets come with a few other state management mechanisms: application state can be maint ained within the ServletContext for the entire web application to use, request-level attributes are useful for sharing information to components handling the same request, and cookies can be placed on the client with configurable expiration times for persistence. For more permanent state storage, servlets usually communicate with databases via JDBC, JPA, or other persistence technologies. Replication and across-container session persistence refers to maint aining session data between container restarts making it an important consideration in enterprise environments, for which most commercial servlet containers provide configurable policy for backups and/or session recovery to ensure high availability. Common security issues in session management are session fixation attacks (which should be prevented by always regenerating session IDs after successful authentication), session hijacking (prevented by setting cookie attributes for same-site, secure and HTTP only and implementing secure HTTPS

communications), cross-site request forgery (CSRF) (solved with synchronizer tokens). Session management strategies are also influenced by performance considerations, where too much session data can lead to bloated memory and eventually impact garbage collection, and session replication in clustered environments can lead to added network overhead. Prevaring on these different state persistence strategies help servlet programmer use best suited options according to specific applicaiton scenario criction optimal usage state action, proformance, Scalability and securtiy.

12.6: Servlet Security and Authentication Mechanisms

The knowledge is based on.java servlet SecurityServlet Security is an important concern for application in servlet such as the security is the general mechanism for securing web resources as well as authentication, authorization and confidentiality of application data. As the servlet container, this is the main enforcement point of those security controls that, together with application defined constraint s, build a strong security architecture. The basic security model in servlet applications is based on the concepts of realms, users, roles, and constraint s. Authentication (authentication mechanisms) defines who you are, while authorization (authorization mechanisms) defines what you can access and action and is based on the given roles of the authenticated user. The deployment descriptor (web. xml), security constraint s are defined using the element , which binds collections of web resources (represented in identified URL patterns) with two constraint s: those of authorization (user roles), and those of transport guarantee (HTTP or HTTPS). NASDAQ: SQ, which provides payments, post-trade risk management, and compliance solutions as well as rich-data research. Modern servlet containers also include additional security features such as HTTP Strict transport security (HSTS), Content security policy (CSP), and an access via OAuth and OpenID Connect for federated authentication scenarios. Security filters are yet another very powerful means to define cross-cutting security concerns (such as input validation, output sanitization and access logging) in one location and have them invoked during the lifecycle of each of your servlets. Aware of these various security measures, developers are empowered with the knowledge necessary to execute defense-in-depth concepts tailored to their application's threat profile

but at the same time protecting servlets applications so sensitive resources and information are never exploited by attackers, while still giving access to the legitimate user. Since security threats on the web keep changing, it is very important to update the web security best practices and use the servlet specification's security features and other protections as per the need to develop and maint ain secure web applications.

(i) Advanced Features and Modern Servlet Capabilities

Since then, the Java Servlet specification has undergone many generations of enhancement and refinement, delivering features that significantly boost developer productivity, achieve better application performance, and provide for more flexible architectures throughout. Servlets have come a long way since their early design, and current implementations are significantly more advanced than just the simple request-response model that served as the backbone of the early web applications. The new annotation-based configuration that comes with Servlet 3.0 radically changes servlet development, eliminating much or all of the necessary XML deployment descriptor dependency. Instead of the traditional xml-based configuration, APIs like @WebServlet, @WebFilter and @WebListener allow for declarative configuration, directly in Java code, which enables simple deployment and better code readability. This approach enables you to configure for URL mapping, initialization parameters, description metadata and other configuration features that were limited to web related. xml. Servlet 3.0 int roduced asynchronous request processing capabilities; these additional capabilities were subsequently enhanced in the next versions to help tackle scalability issues from long-running operations by releasing a thread during processing. This allows servlets to start async operations that could take a long time to complete without holding up container threads, and can help applications become more responsive to incoming requests and use system resources more efficiently under load. The API supports container-managed asynchronous processing and application-managed threading, with mechanisms for timeout handling and completion notification. Servlet 3.1: As the first major specification after Servlet 3.0, Servlet 3.1 int roduced Non-blocking I/O support, which allowing Servlet vendors to implement a scale-out model for improved scalability by allowing

asynchronous reading and writing of request and response data. This ability to react to events as they occur, rather than waiting for all the elements to be present at once can be important when uploading or downloading files over the wire, as well as processing streamed data or int egrating with reactive programming models. To support such db functions, both the ReadListener and WriteListener int erfaces enable it to send notifications to its applications where data can be read, or it can write the output buffer which is empty and its post data isn't blocked. Servlet 40 enables HTTP/2 supported servlets through which servlets can take advantage of performance char acteristics offered by the new protocol version, such as multiplexing, header compression and server push features. PushBuilder provides an API that allows for server push, where the Servlet can send resources to the client out of band, before the client has even requested them. This includes stochastic servlets, filters, and listeners for web applications, enabling application initialization code to programmatically attach them instead of a static declaration in the web.xml file. This breaks apart a glass wall and builds flexible structures between web applications and frameworks. Subsequent servlet versions int roduced embedded container capabilities, allowing applications to programmatically configure and launch servlet containers themselves, supporting microservice architectures and simpler deployment models. Fragment web. This is similar to the built-in ability of Spring to provide extension point s that the libraries can also contribute to when configuring the web applications, and the web.xml support allows libraries to contribute as well. The other important concept is the ServletContainerInitializer mechanism through which library authors can add hooks to the framework initialization by configuring information on int egration point s in their last descriptor. Security lattice across versions is having great features like, programmatic authentication, role mapping, and int egration with Java EE/Jakarta EE security frameworks. Same with multipart request handler allows you to process file uploads, if API is common then all multipart requests will be parsed in the same way and protocol upgrade support allows you transition from HTTP to WebSocket or similar protocols. As for security, we use JSR-375 (Java EE Security API) int egration, which gives us the latest security practices from identity stores to authentication mechanims to security context concerns. Together, they

facilitate modern web development yet retain compatibility with existing code bases. By recognizing and harnessing these capabilities, developers can create advanced, high-performing web applications that align with contemporary demands for responsiveness, scalability, and developer productivity, thereby ensuring that servlet technology retains its relevance in the modern software development landscape, despite the rise of alternative frameworks and architectural approaches.

12.7: Integration with Java EE/Jakarta EE and Ecosystem Considerations

Java Servlets are a part of the larger Java EE (now Jakarta EE) ecosystem, providing a building block technology that int eracts with many other specifications and frameworks to build complete enterprise applications. So, in-development systems, where servlets work, quite int egrates well int o those convolution landscapes. At the specification level, servlets work closely with many Java EE technologies, including JavaServer Pages (JSP), which is a view technology that compiles int o servlets behind the curtains; Expression Language (EL), which provides a clean syntax for accessing data within JSP pages and other templating technologies; the JSP Standard Tag Library (JSTL) to extend JSP functionality with reusable tag components; and lastly, JavaServer Faces (JSF), which builds a component-based UI framework on top of the servlet foundation. The servlet container also implements a number of Java EE specifications other than servlets such as JNDI (Java Naming and Directory Int erface) for resource lookups, JDBC (Java Database Connectivity) for database access, JTA (Java Transaction API) for transaction management, JMS (Java Message Service) for messaging, and various security technologies like JAAS (Java Authentication and Authorization Service). Such a container environment helps servlets access them through standard APIs, readily available data sources include DataSources, JMS destinations, and EJBs (Enterprise JavaBeans) via JNDI lookups or injection mechanisms. In modern servlet environments (Java Servlets, Java EE, Jakarta EE, etc.), dependency injection happens with CDI (Contexts and Dependency Injection), which is the type-safe, extensible way to access a resource/component. Then, through annotations like @Inject (along with producer methods and qualifiers), servlets can get their dependencies as injected without any code to look them up manually.

Bean Validation with Servlets Bean Validation enables declarative validation of request parameters and form submissions. Servlets can leverage a number of frameworks and libraries beyond servlet technology itself: persistence technologies such as JPA (Java Persistence API), Hibernate, or MyBatis; web frameworks such as Spring MVC, Struts or Play Framework (most are designed on top of servlet technology); template engines such as Thymeleaf, FreeMarker, or Velocity; and utility libraries for JSON processing, XML, logging, and other cross-cutting concerns. The microservices architectural trend impacted how servlets are deployed, as frameworks such as Spring Boot, WildFly Swarm/Thorntail, and Payara Micro allow for the serving of self-contained applications with embedded servlet containers. These cloud deployment factors influence servlet applications via Docker containerization, Kubernetes orchestration and int egration of cloud services. In servlet-based environments, performance-enhancing techniques include connection pooling, in-memory and distributed caching strategies, distributing loads among several containers and resource management. To test servlet applications, you have specialized frameworks like JUnit, Mockito, Spring Test, Arquillian, and tools that simulate HTTP requests. There are also namespace changes because of the transition from Java EE to Jakarta EE (from javax. * to jakarta. *) and governance changes but the core int egration archetypes remain unchanged. Technologies such as Jakarta EE Faces Flow and Security, MicroProfile for microservices development and GraalVM native image compilation will continue to evolve the ecosystem around reactive programming models, better application microservice development and consumption in startup time and resources. By being aware of these int egration point s and ecosystem considerations, developers can make informed architectural decisions, choose the right technologies for the different needs of their application, and build servlet-based applications that take full advantage of the rich features and services offered by the Java enterprise platform as a whole.

12.8: Servlet Life Cycle: Stages in Servlet Execution

The servlet life cycle is one of the basic concepts of Java web development, indicating the specific order of actions that take place between the instantiation and finalization of the servlet. Servlets differ

from regular Java applications in that there is no well-defined main method that serves as their entry point ; they run inside the managed environment of a servlet container (for example, Apache Tomcat, Jetty or JBoss) which takes responsibility for handling the lifecycle of servlet instances by instantiating, initializing, invoking, and finally destroying servlet instances according to a specified protocol. This lifecycle is vitally important for Java developers who are creating enterprise web applications, as it gives a roadmap of how to manage HTTP requests properly while allowing for appropriate resource management, resulting in the application working smoothly during its time running. Now, servlets go through the following phases: loading and instantiation, initialization, service processing (request handling, response generation), and destruction. So, these stages serves for a specific purpose and they provides developers with hooks to implement specific behavior through methods that are defined in the javax. servlet. Servlet int erface. In this Unit, we will take a closer look at these stages and the evolution of servlets in terms of their purpose, details on how they work and the proper techniques to handling the execution process in enjoyable Java Web applications. Understanding servlet life cycle empowers developers to build not only powerful but highly efficient and scalable web applications that manage resources effectively, handle concurrent requests, and implement complex business logic while adhering to the separation of concerns principle that is a cornerstone of modern software architecture.

(i) Loading and Instantiation Phase

The loading and instantiation phase is the first phase in the servlet life cycle, during which the servlet container is first notified of a servlet and loads the servlet int o an execution environment. A servlet class is loaded usually at one of the 3 moments in time: at container startup, during first request of the servlet, or at an explicit time, defined in the deployment descriptor (web. xml) or through annotations. When the servlet container is initialized, it looks at the web application's configuration files, most notably the deployment descriptor (web. xml) or servlet annotations in the case of modern applications—would indicate servlets to be loaded on startup, by marking these servlets with a element in web. xml or by using the loadOnStartup attribute of the @WebServlet annotation in code. These elements take int egers

representing the relative order in which servlets are to be initialized, with smaller numbers receiving higher priority; negative values (or the absence of the element) signify that the servlet is to be loaded only on its first request. Now when the container finds the servlet class , it loads the servlet class int o memory using the Java Class Loader also making sure that the class es and the libraries required by the class are available in the class path. And only after loading successfully this container calls its no-argument constructor of the servlet, which is an instance we will use for dealing with all the requests for the application, keeping in mind that is actually a singleton in relation to the servlet context.

Understanding this instantiation mechanism is crucial for developers to implement servlets correctly according to certain rules. First, servlet class es must implement a public no-argument constructor, since the container uses reflection to create instances without passing parameters. This constructor should remain lightweight and should not contain complex initialization logic: proper initialization will need to be deferred until the initialization phase discussed in the upcoming section. Second, a single servlet instance is used to process multiple requests, and they might come at the same time, so instance variables should be used with caution because this can be a thread safety issue — otherwise it is better to use immutable objects or thread-local storage to maint ain state between method invocations. It must implement the javax. servlet. Servlet int erface, usually by sub-class ing the javax. servlet. GenericServlet class for protocol-independent servlets or the javax. servlet. http. HttpServlet A class for HTTP-specific servlets, which is a base class that provides default implementations of the int erface methods. The servlet context is selected too at the time of instantiation, allowing the servlet to be served with access to the application-wide ServletContext that gives it access to key elements of configuration, keys for parameters, and connectors for applications that allow for int er-app communication across the application. This context allows servlets to share information among themselves, read configuration parameters, and int eract with other components of the web application. This loading and instantiation step culminates in the servlet instance being created (but not yet ready to be called), ready for an initialization step. This phase is mainly based on activities managed by the container, with little developer int ervention, but knowing how

it works under the hood is important so that you can implement the design of your servlets in such a way that they work well in the container environment, especially when you implement custom class loading or need to work with complex dependency scenarios.

(ii) Initialization Phase

The Initialization Phase signifies the servlet moves away from just being an instance of a class to an entity that can actually serve requests. So you know this important moment occurs right after instantiation, when the servlet container invokes the servlet's init(ServletConfig config) method, a contract method defined in the javax. servlet. All servlets must implement this int erface, which is a Servlet int erface. The key objective of this step is to provide an opportunity for the servlet to initialize one-time setup stuff (like loading configuration values, getting database connections, creating resource pools, etc.) that will be used during the full lifecycle of the servlet. A ServletConfig object is passed to the init() method by the container — this object allows the servlet to access configuration parameters specified in the deployment descriptor(web. xml) or through annotations. This object acts as a middle ground between the deployment configuration and the servlet code itself enabling the developer to extract configuration details away from the code, thus changing behavior without changing code. In addition, the servlet can get a reference to the ServletContext object that represents the web application and can be used to get access to application-wide resources and functionality via the ServletConfig. It is contra the event if the init() method is invoked at least once in the life cycle of a servlet, so controlling the initialization that never repeats if once the servlet instance will initialize. Because the initialization phase offers a precious opportunity to create resource-int ensive setup tasks that can be then amortized to all the following request processing, since many requests may be ultimately processed by this one servlet instance.

If initialization fails the init() method throws a ServletException, allowing servlets to signal serious errors that preclude their functioning. By doing so, it ensures that the servlet doesn't get int o action in a half-baked or bad state, which can lead to erratic application behavior or even expose a servlet to security threats. Initialization tasks can include opening database connections, creating connection pools,

initializing caching mechanisms, loading configuration files, establishing a network connection to a remote service, precomputing results, and constructing data structures that are used to support the servlet's primary function. Because the init() method is only called once, developers need to make sure that all necessary resources are acquired and configured correctly at this stage, with suitable error handling in place so that initialization errors can be handled gracefully. The GenericServlet abstract class implements a default version of the init(ServletConfig) method, which stores the config object and then calls a no-argument init() method that subclass es can override to implement their initialization logic without needing to manage the ServletConfig reference that will be stored for them. The configuration management separation pattern helps in the development of servlets by allowing the configuration management logic to be separate from the specific business logic implementation. Different approaches can be taken in the initialization phase to gear up for an application, e.g., lazy initialization of expensive resources or eager initialization of critical components, based on the performance needs and resource limits of the application. The initialization phase ends when the servlet goes int o the service phase awaiting requests from the client.

(iii) Service Phase - Request Processing

Servlet Life Cycle The service methodServlet Life Cycle--The Service Phase. This step starts when the servlet container receives an appropriate HTTP request and invokes the servlet's service(ServletRequest req, ServletResponse res) method, which details the request and a channel to build the response. For HTTP servlets (the most usual species in modern web applications), the container actually invokes the service(HttpServletRequest req, HttpServletResponse res) method of the HttpServlet class , which receives HTTP-specific request and response objects populated with protocol-relevant information. The default implementation of this method given by HttpServlet checks the HTTP method (GET, POST, PUT, DELETE… etc.) and calls the relevant method of the servlet: doGet(), doPost(), doPut(), doDelete()… etc. The pattern of delegation simplifies the servlet development because developers only need to implement the methods representing the HTTP methods the application supports and not handle the dispatching themselves. All of those

method specific handlers receive identical request and response objects that allows them to inspect any request params, any request headers, and request content and to produce appropriate responses, including status codes, response headers, and response body content. The service phase, unlike initialization and destruction, occurs during the lifetime of the servlet and will be executed whenever a request is made to the servlet, either once or multiple times, on different threads, to handle multiple requests.

Handling requests in a multi-threaded manner is both a performance gain and a huge development hurdle. Handling concurrent requests efficiently without creating a new request thread per client per request is typically accomplished by the servlet container (e.g., Tomcat) by means of a request context (thread pool) that it manages under the covers. This model enables a single instance of a servlet to handle multiple clients simultaneously, thereby significantly improving scalability compared to creating a separate instance per client. But this shared-instance model makes it important to focus on thread safety, because instance variable is shared across all service method invocations. To avoid the inevitable pitfalls of managing state in this environment, there are a few strategies: we can use synchronization to guard shared resources, we can use thread-local storage to store request specific data, the local variables that are scoped to the thread's stack, we can use immutable objects that we can pass around safely, or we can use session mechanisms to hold client specific state. It also includes important processing steps that developers have to implement, e.g. parsing request parameters and headers, mariage or authenticate the user if applicable, apply the application-specific business logic and formulate a fitting response, which covers the status code as well as headers and content. Service phase: Handling errors is pivotal, exceptions should be caught and converted int o HTTP compatible error representations. Also, the servlet API allows request dispatching between servlets, which is useful for maint ainability, permission checking, and modularity. During the service phase, performance considerations of minimizing processing time, good memory usage, resource management and caching of frequently used data or computations to decrease response time come int o account. We can say that the service phase exists for as long as the servlet is running, handling requests until the servlet container calls the destroy phase.

(iv) Service Phase - Response Generation

After processing the request during the service phase, servlets need to create and send appropriate responses back to clients, thus completing the request-response loop at the core of HTTP int eraction. For the response generation, we use (and are given) the HttpServletResponse object provided by the container, which includes methods to set status codes, headers, content type, get output streams or writers to send the response body, etc. Status codes convey the result of processing requests—for example, 200 (OK) if a request was processed successfully, 404 (Not Found) if it tried to access resources that don't exist, or 500 (Int ernal Server Error) if the server failed to handle the request—and should always be configured before writing any response content. HTTP headers are used to send additional information about the data being transmitted along with the response, and they control how caching should work, attributes of the transport layer, security rules, and many other things between the client and server; for example, common headers include Content-Type, Content-Length, Cache-Control, and Set-Cookie. Using the setContentType() method, the response content type, or the format of the data (text/html, application/json, image/jpeg, etc), and the char   acter encoding for textual content is included on the response to help the client correctly parse and render the response data. Servlets can generate the response body with either a Print Writer (obtained by calling getWriter()) (for char   acter data), or another type of output stream (obtained by calling getOutputStream()) (for binary data), but not both within the same response (as this constitutes a violation of the servlet specification and results in an IllegalStateException being thrown). The response generation technique largely depends on type and nature of the application. Servlets directly create markup using print statements or use template engines like Thymeleaf or FreeMarker to separate presentation logic from business logic for HTML based applications, which will delegate rendering JSP (JavaServer Pages) using request dispatch. In data-centric applications, servlets typically return JSON or XML payloads, leveraging libraries such as Jackson, Gson or JAXB to marshal/unmarshal Java beans to/from these serialized representations. Binary data lowers content—like PDF, images, or downloadable files—requires some business and particular consideration, including content kind, content disposition headers, and safe streaming strategies

to handle massive files effectively. Complex response patterns are now common in web applications, such as partial updates for AJAX-based int erfaces, streaming for large data sets or real-time updates, compression to reduce the bandwidth footprint , and content negotiation to return different representations depending on what the clients can or want. Caching directives are another important part of response generation, allowing servlets to hint to clients and int ermediaries about whether contents are fresh and reusable, reducing load and improving performance. Likewise, the ability to manage cookies via the Cookie class and the addCookie() method allows servers to track sessions and maint ain stateful int eractions through the inherently stateless HTTP protocol. Error handling during response generation needs to be treated differently, since exceptions raised after part of the response was delivered can cause corrupted or partial content to be delivered; typically proper error handling involves both buffer management and error pages mapped in the deployment descriptor. Once the response has been generated, the servlet container takes care of the underlying work of sending the response back over the network connection and getting ready for the next request. While generating the response, servlet must be aware of performance implications, such as memory consumed when generating large response, buffered output to tradeoff between memory usage and responsive, and freeing resource associated with response to avoid leak when operating in high volume.

(v) Destruction Phase

Although the destruction phase marks the last stage in the servlet life cycle, it takes place when a servlet needs to be taken out of service by the servlet container. This phase is invoked under a variety of situations, such as when the web application is being undeployed or redeployed or if the servlet container is shutting down gracefully, or when the container needs to recover resources. The service phase will be invoked thousands or millions of times in the lifetime of the servlet (and will also be executed on a separate thread for each request), but the destruction phase will be executed (like the initialization phase) a guaranteed — exactly once — for each servlet instance that is created. This container indicates the start of this phase by invoking the destroy() method on the servlet, which is a contract method written in javax.

servlet. This int erface allows servlets to be given the chance to do some cleanup work when the servlet is being taken out of service. The destroy() method is primarily responsible for releasing resources — closing database connections, terminating network connections, shutting down thread pools, releasing file handles, and freeing any other system resources that the initialized acquired during the initialization phase or in the servlet's operational life. The cleanup also prevents resource leaks that might exist beyond the servlet lifetime, eventually leading to performance degradation or server instability. Moreover, the destruction phase allows you to persist state information that must outlive the current application instance (for example, saving accumulated statistics, unsaved data, or configuration changes to permanent storage).

During the destruction phase, the servlet container guarantees a graceful shutdown. It guarantees that before calling destroy() all the threads currently running in the service method must complete their processing or are given a reasonable time to do so. That is, the destroy() method will not fire until ALL service method invocations have exited or a container-specific timeout has occurred. After calling destroy(),

(vi) Concurrency and Thread Safety

Managing concurrency is one of the biggest challenges in servlet development because a servlet by design pattern, is a single instance that is invoked by multiple clients (in parallel). Instead of the common approach in programming model where each client request receives its own application instance, the servlet container follows the singleton approach with multi-threaded execution, leading to a shared application space with the necessity of mitigating the risks associated with shared state in a multi-threaded environment. Hint : When a servlet container (like an application server or web server) handles multiple concurrent requests directed at the same servlet, it may forward these requests in parallel by calling the servlet's service() method in separate threads. The potentially huge performance advantages derived from this concurrency model comes at the expense of a shared state with respect to instance variables (fields) of the servlet as it is instantiated per application rather than per request. Therefore, at the servlet instance level, every instance variable is at risk of race conditions, data corruption, and other concurrency problems unless appropriately

safeguarded. Concurrency can be handled in servlets in four ways: making your servlet thread-safe by synchronizing yourself with your critical section code or maint aining an immutable state, using local variables instead of instance variables — since local variables are thread-local automatically as they are created on the stack of the thread, using the thread-local storage pattern or ThreadLocal class to persist thread-specific state, or using an int erface known as SingleThreadModel (which has been deprecated for now in at least the last couple of servlet specifications) which allows the web container to enforce that only one thread accesses a servlet instance at any time, so that the container needs to keep a pool of servlet instances.

For servlets that need to retain state between requests, certain concurrent programming strategies are helpful. Synchronization is the most simple solution to the problem, using Java's synchronized keyword or explicit locks provided in java. util. concurrent. ensure that only a single thread can execute a specific part of code or a shared resource at a time and can be found in the locks package. Synchronization, on the other hand, comes with performance overhead in the forms of thread contention and possible deadlocks and so is only appropriate for short-lived, sparse operations. Many concurrency scenarios can be elegantly addressed using immutable objects [Java Concurrency] which can safely be shared across threads, without synchronization, after they've been built; this is the case for things like configuration data or pre-computed results that will not change during servlet execution. Java.Core. Concurrent.Collections class es util. implementations in the java.util.concurrent package — ConcurrentHashMap, CopyOnWriteArrayList, and BlockingQueue implementations, for example—provide thread-safe alternatives to the standard collections with better performance char acteristics than explicitly synchronized collections. The HttpSession API is designed to handle user-specific state by maint aining a container-managed thread-safe association between data and a particular client session instead of relying on a servlet to do so, which further delegates the thread-safety concern to the container. Other than these basic techniques, servlets supporting significant concurrent traffic will often use more advanced patterns like the read-write lock pattern for resources that are expensive to acquire but that are heavily read and seldom written, double-check locking for lazily initialized expensive

resources, or compare-and-swap operations provided by atomic classes such as AtomicInt eger and AtomicReference lock-free updates to trivial values. Testing servlets for thread safety is particularly difficult, needing specialized tools such as stress testing frameworks, static analysis tools to catch possible concurrency issues or explicit concurrency testing frameworks that can generate managed race conditions. Servlet concurrency can be achieved by following certain principles and practices in your application development lifecycle.

(vii) Advanced Life Cycle Considerations

In addition to the basic lifecycles stages, there are various advanced aspects that heavily influence servlet functioning and efficiency in real-world applications. Servlet initialization parameters is a method of configuring servlets without changing code so the deployment can set it to whatever it wants. These parameters can be defined through the element in web. xml or the initParams field of the @WebServlet annotation and accessed during initialization phase via the ServletConfig. getInitParameter() method. This configuration construct fosters the separation of code from configuration, allowing the same piece of servlet code to run differently on different environments. Load-on-startup settings dictate exactly when servlet initialization happens, optimizing startup time and request latency. Servlets with positive int egers in their element or loadOnStartup annotation attribute are constructed at container startup in increasing numerical order so that critical servlets are in place when the application first receives traffic, while servlets that lack this directive or have negative values construct lazy on first request. This helps a lot for servlets which have expensive initialization processes, and the servlets provides low-level or any services which are required by other components. Error handling is another high-level concept that straddles the servlet life cycle, including both programmatic exception handling in servlet methods and declaratively defined error page mappings in the deployment descriptor, which direct specific types of exception or HTTP error codes to dedicated error-handling servlets or JSP pages, thereby allowing for consistent error presentation across the application while allowing for generic information to be hidden that a developer can use to troubleshoot.

Servlet context listeners allow for the management of an application's life cycle, passing the life cycle management from individual servlets to the application level, by implementing the ServletContextListener int erface and being notified of an applications startup and shutdown through the contextInitialized() and contextDestroyed() methods respectively. These listeners usually make application-wide initializations and clean-ups like creating database connection pools, logging configuration, Caches preloading, JDBC drivers registration, In a similar way, session listeners — that is, class es that implement the HttpSessionListener, HttpSessionAttributeListener, and HttpSessionBindingListener int erfaces — allow code to be executed when a session is created, destroyed, and when its attributes change: useful for keeping track of users, managing resources, and for security monitoring purposes. The asynchronous processing, which was int roduced in the Servlet 3.0 specification, changes the conventional request-response life cycle by letting servlets perform long-running operations while freeing the container's request-processing thread. By calling request. Either doAllInOneThread() of startAsync() method in some servlet, where the servlet get the AsyncContext object, which disassociates the request and response object from the current thread, allowing original thread for returning to the container's thread pool while processing is continued on another thread, and may be end much later. This pattern is useful for long-running operations, server-push technologies, and non-blocking I/O system int egration. The servlet specification additionally defines resource management through annotations including @Resource, @Resources, and @PostConstruct/@PreDestroy, which enables resource injection and life cycle method designation that incorporates with the container's higher resource management amenities.

Production deployments add a number of other life cycle considerations. For example, many containers will be able to reload servlets, so that if a servlet class changes, the servlet can be detected and getting going through the life cycle of destruction and initialization without a restart of the application, which can be useful during development but can sometimes be turned off in production for performance reasons. One of the special challenges with clustering environments, where different physical or virtual machines may run multiple servlet containers: session replication, distributed caching,

synchronized initialization have to be considered because servlets aren't singletons, and their life cycle management should be handled specifically. While supplying a servlet involves specifying which bytecode will be executed, there are security concerns that int ersect the servlet life cycle that you need to include in your design, including role-based access controls that restrict which users can access which servlets, programmatic security checks that you perform in servlet methods, and secure initialization that protects sensitive configuration data. Lifecycle performance tuning through connection pooling at initialization, request dispatching during the service phase, response caching between requests, and resource management at destruction. Most servlet life cycle monitoring and debugging relies on recording important transitions int o a log file, container-specific utilities (such as the one that tracks servlet life cycle and state), or JMX (Java Management Extensions), exposing servlet metrics and state data to outboard monitoring systems. Having advanced the understanding of the servlets life cycle, developers can ideally design servlets for correctness, efficiency, scalability, enterprise int egration, and optimal operation in hard times.

With Java EE, a specified life cycle for the servlet engines gives a structured life cycle framework — char acterized in tall levels below. By thoroughly understanding and appropriately utilizing the functionalities of each phase, developers are able to craft resilient, performant, and maint ainable web applications that make effective use of the servlet container's services and adhere to correct resource management and concurrency control protocols. However, a deep understanding of servlet life cycle is always essential to Java web application development regardless of being simple web applications or complex enterprise solutions.

12.9: Reading Form Data from Servlet

Enter user input One of the most basic tasks that are performed in web applications. [To know more JAVA SERVLETS] – How To Handle HTML Forms In Servlets? Knowing how to retrieve, validate and then use this data effectively is one of the key parts of building int eractive web applications.

(i) Understanding HTTP Form Submission

So, when someone fills a form on a webpage, the data is sent to a server with an HTTP request. The form data can be sent in one of two ways, depending on how the form is configured:

GET Method: The form data is added to the URL as a query String parameter. This is usable with non-sensitive data and when you may want to bookmark the outcome.

POST Method: As the form data is sent as part of the HTTP request body, it is not visible in the URL. Sensitive information, large amounts of data, or a request that might mutate server state should be passed via the body by this method The HTML markup for these form types looks like this:

```
<!-- GET method form -->
<form action="processForm" method="get">
<input type="text" name="username">
<input type="submit" value="Submit">
</form>
<!-- POST method form -->
<form action="processForm" method="post">
<input type="password" name="password">
<input type="submit" value="Submit">
</form>
```

(ii) Extracting Form Data in Servlets

There are some methods in Java servlets for extracting the form data. The main methods are defined in the HttpServletRequest int erface, and are slightly different depending on whether the data was submitted using GET or POST.

Basic Parameter Retrieval -getParameter(String name) is the most common and used method which receives the parameter name and returns the value associated with the parameter name as a String

```
@WebServlet("/processForm")
public     class  FormProcessorServlet extends HttpServlet {
   protected void    doGet(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
    // Retrieve a single parameter value
    String   username = request.getParameter("username");
```

```
    // Process the username
    if (username != null && !username.isEmpty()) {
       // Valid username provided
       response.getWriter().print ln("Hello, " + username + "!");
    } else {
       // No username or empty username
       response.getWriter().print ln("Hello, guest!");
    }
  }
  protected void    doPost(HttpServletRequest request,
HttpServletResponse response)
       throws ServletException, IOException {
    // For POST requests, we can use the same getParameter method
    doGet(request, response);
  }
}
```

Handling Multiple Values

We use getParameterValues(String   name) to get multiple values (when we have checkbox or multi-select list in our form, with the same name) as a String   array.:

```
@WebServlet("/processInt erests")
public        class  Int erestProcessorServlet extends HttpServlet {
  protected void    doPost(HttpServletRequest request,
HttpServletResponse response)
       throws ServletException, IOException {
    // Retrieve multiple values for the same parameter
    String  [] int erests = request.getParameterValues("int erest");
    response.setContentType("text/html");
    Print Writer out = response.getWriter();
    out.print ln("<html><body>");
    out.print ln("<h2>Your Selected Int erests:</h2>");
    if (int erests != null && int erests.length > 0) {
       out.print ln("<ul>");
       for (String   int erest : int erests) {
          out.print ln("<li>" + int erest + "</li>");
       }
       out.print ln("</ul>");
```

```
        } else {
            out.print ln("<p>No int erests selected.</p>");
        }
        out.print ln("</body></html>");
    }
}
```

Retrieving All Parameters

To retrieve all the parameters that were passed in with a form, use getParameterNames() to obtain an enumeration of the parameter names, then iterate through them to get the values of each parameter:

```
@WebServlet("/displayAllParams")
public      class ParameterDisplayServlet extends HttpServlet {
    protected void     doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html");
        Print Writer out = response.getWriter();
        out.print ln("<html><body>");
        out.print ln("<h2>All Form Parameters:</h2>");
        out.print ln("<table border='1'>");
        out.print ln("<tr><th>Parameter Name</th><th>Parameter Value(s)</th></tr>");
        Enumeration<String  > paramNames = request.getParameterNames();
        while (paramNames.hasMoreElements()) {
            String   paramName = paramNames.nextElement();
            out.print ln("<tr><td>" + paramName + "</td><td>");
            String  [] paramValues = request.getParameterValues(paramName);
            if (paramValues.length == 1) {
                String   paramValue = paramValues[0];
                if (paramValue.length() == 0) {
                    out.print ln("<i>No Value</i>");
                } else {
                    out.print ln(paramValue);
                }
            } else {
                out.print ln("<ul>");
```

```
        for (String   paramValue : paramValues) {
            out.print ln("<li>" + paramValue + "</li>");
        }
        out.print ln("</ul>");
    }
    out.print ln("</td></tr>");
}
out.print ln("</table>");
out.print ln("</body></html>");
}
    protected void    doPost(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
    doGet(request, response);
    }
}
```

Using the Parameter Map

For more structured parameter handling, getParameterMap() returns a Map containing parameter names as keys and parameter values as String   arrays:

```
@WebServlet("/processMapForm")
public       class  ParameterMapServlet extends HttpServlet {
    protected void    doPost(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
    Map<String  , String  []> parameterMap =
request.getParameterMap();
    response.setContentType("text/html");
    Print Writer out = response.getWriter();
    out.print ln("<html><body>");
    out.print ln("<h2>Form Data Summary:</h2>");
    // Process all parameters using the map
    for (Map.Entry<String  , String  []> entry :
parameterMap.entrySet()) {
        String   paramName = entry.getKey();
        String  [] paramValues = entry.getValue();
        out.print ln("<p><strong>" + paramName + ":</strong> ");
        if (paramValues.length == 1) {
```

```
        out.print ln(paramValues[0]);
      } else {
        out.print ln("<br>");
        for (String   value : paramValues) {
          out.print ln("- " + value + "<br>");
        }
      }

      out.print ln("</p>");
    }
    out.print ln("</body></html>");
  }
}
```

(iii) Char   acter Encoding Considerations

To retrieve all the parameters that were passed in with a form, use getParameterNames() to obtain an enumeration of the parameter names, then iterate through them to get the values of each parameter:

```
@WebServlet("/int ernationalForm")
public      class  Int ernationalFormServlet extends HttpServlet {
  protected void    doPost(HttpServletRequest request,
HttpServletResponse response)
      throws ServletException, IOException {
    // Set char   acter encoding before retrieving parameters
    request.setChar   acterEncoding("UTF-8");
    // Now retrieve parameters with proper encoding
    String   name = request.getParameter("name");
    String   address = request.getParameter("address");
    // Set response encoding
    response.setContentType("text/html; char   set=UTF-8");
    Print Writer out = response.getWriter();
    out.print ln("<html><body>");
    out.print ln("<h2>Int ernational Form Data:</h2>");
    out.print ln("<p>Name: " + name + "</p>");
    out.print ln("<p>Address: " + address + "</p>");
    out.print ln("</body></html>");
  }
}
```

(iv) Processing Different Form Data Types

Form data is always transmitted as String s, but your application may need to convert these String s to appropriate data types for processing.

Type Conversion

If you want to handle the parameters in a more structured way, you can use getParameterMap(): It returns a Map that has parameter names as keys and parameter values as String arrays:

```java
@WebServlet("/calculateTotal")
public      class ShoppingCartServlet extends HttpServlet {
   protected void    doPost(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
    try {
       // Convert String   to int eger
       int  quantity = Int eger.parseInt
(request.getParameter("quantity"));
       // Convert String   to double
       double price =
Double.parseDouble(request.getParameter("price"));
      // Convert String   to boolean
      boolean isGift =
Boolean.parseBoolean(request.getParameter("gift"));
      // Perform calculations
      double total = quantity * price;
      if (isGift) {
         total += 5.00; // Gift wrapping fee
      }
      response.setContentType("text/html");
      Print Writer out = response.getWriter();
      out.print ln("<html><body>");
      out.print ln("<h2>Order Summary</h2>");
      out.print ln("<p>Quantity: " + quantity + "</p>");
      out.print ln("<p>Price per unit: $" + String  .format("%.2f",
price) + "</p>");
      out.print ln("<p>Gift wrapping: " + (isGift ? "Yes" : "No") +
"</p>");
```

```
        out.print ln("<p>Total: $" + String  .format("%.2f", total) +
"</p>");
        out.print ln("</body></html>");


    } catch (NumberFormatException e) {
        // Handle parsing errors
       response.sendError(HttpServletResponse.SC_BAD_REQUEST,
"Invalid number format in form data");
    }
  }
}
```

Handling Date Inputs

Converting String   date inputs to java.util.Date objects:

```
@WebServlet("/processDate")
public      class  DateProcessorServlet extends HttpServlet {
   protected void    doPost(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
     String   dateString  = request.getParameter("eventDate");
     response.setContentType("text/html");
     Print Writer out = response.getWriter();
     try {
        SimpleDateFormat dateFormat = new
SimpleDateFormat("yyyy-MM-dd");
        Date eventDate = dateFormat.parse(dateString  );
        // Calculate days until event
        long daysDiff = (eventDate.getTime() - new
Date().getTime()) / (1000 * 60 * 60 * 24);
        out.print ln("<html><body>");
        out.print ln("<h2>Event Information</h2>");
        out.print ln("<p>Event Date: " +
dateFormat.format(eventDate) + "</p>");
        out.print ln("<p>Days until event: " + daysDiff + "</p>");
        out.print ln("</body></html>");
     } catch (ParseException e) {
        out.print ln("<html><body>");
        out.print ln("<h2>Error</h2>");
```

```
        out.print ln("<p>Invalid date format. Please use yyyy-MM-dd
format.</p>");
        out.print ln("</body></html>");
    }
  }
}
```

(v) Handling File Uploads

For flowing files, the getParameter() methods of the standard are not enough. Instead, you must refer to the Part API added in Servlet 3.0 or third-party library such as Apache Commons FileUpload.

Using Servlet 3.0 Part API

```
@WebServlet("/fileUpload")
@MultipartConfig(
   fileSizeThreshold = 1024 * 1024,     // 1 MB
   maxFileSize = 1024 * 1024 * 10,      // 10 MB
   maxRequestSize = 1024 * 1024 * 50)   // 50 MB
public      class  FileUploadServlet extends HttpServlet {
   protected void    doPost(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
     // Get the file part from the request
     Part filePart = request.getPart("file");
     // Extract file information
     String   fileName = getSubmittedFileName(filePart);
     long fileSize = filePart.getSize();
     String   contentType = filePart.getContentType();
     // Define the location to save the file
     String   uploadPath =
getServletContext().getRealPath("/uploads");
     File uploadDir = new      File(uploadPath);
     if (!uploadDir.exists()) {
        uploadDir.mkdir();
     }
     // Save the file
     filePart.write(uploadPath + File.separator + fileName);
     // Process other form fields
     String   description = request.getParameter("description");
```

```
            response.setContentType("text/html");
            Print Writer out = response.getWriter();
            out.print ln("<html><body>");
            out.print ln("<h2>File Upload Summary</h2>");
            out.print ln("<p>File Name: " + fileName + "</p>");
            out.print ln("<p>File Size: " + fileSize + " bytes</p>");
            out.print ln("<p>Content Type: " + contentType + "</p>");
            out.print ln("<p>Description: " + description + "</p>");
            out.print ln("<p>File saved successfully to: " + uploadPath +
"</p>");
            out.print ln("</body></html>");
        }
    // Helper method to extract the file name from the Part header
    private String   getSubmittedFileName(Part part) {
        String   contentDisp = part.getHeader("content-disposition");
        String  [] items = contentDisp.split(";");
        for (String   item : items) {
            if (item.trim().startsWith("filename")) {
                return item.subString  (item.indexOf("=") + 2, item.length()
- 1);
            }
        }
        return "";
    }
}
```

Form Data Validation

Always put security first when dealing with form data. Here are some crucial security practices:

```
@WebServlet("/registerUser")
public       class  UserRegistrationServlet extends HttpServlet {
    protected void     doPost(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {

        String   username = request.getParameter("username");
        String   email = request.getParameter("email");
        String   password = request.getParameter("password");
```

```
    String   confirmPassword =
request.getParameter("confirmPassword");

    List<String > errors = new      ArrayList<>();

    // Validate username
    if (username == null || username.trim().length() < 3) {
        errors.add("Username must be at least 3 char   acters long");
    }

    // Validate email
    if (email == null || !email.matches("^[\\w-\\.]+@([\\w-]+\\.)+[\\w-
]{2,4}$")) {
        errors.add("Please enter a valid email address");
    }

    // Validate password
    if (password == null || password.length() < 8) {
        errors.add("Password must be at least 8 char   acters long");
    }

    // Confirm passwords match
    if (!password.equals(confirmPassword)) {
        errors.add("Passwords do not match");
    }

    response.setContentType("text/html");
    Print Writer out = response.getWriter();

    out.print ln("<html><body>");

    if (errors.isEmpty()) {
        // All validations passed, process the registration
        out.print ln("<h2>Registration Successful</h2>");
        out.print ln("<p>Username: " + username + "</p>");
        out.print ln("<p>Email: " + email + "</p>");
        // In a real application, you would save the user to a database
here
```

```
        } else {
          // Validation errors found
          out.print ln("<h2>Registration Failed</h2>");
          out.print ln("<p>Please correct the following errors:</p>");
          out.print ln("<ul>");
          for (String   error : errors) {
              out.print ln("<li>" + error + "</li>");
          }
          out.print ln("</ul>");
          out.print ln("<p><a href='javascript:history.back()'>Go back
and try again</a></p>");
        }

      out.print ln("</body></html>");
   }
}
```

(vi) Security Considerations

Always put security first when dealing with form data. Here are
some crucial security practices:

Input Sanitization

Note: Always sanitize user input to avoid   security problems such
as XSS attacks:

```
@WebServlet("/commentProcess")
public      class  CommentProcessorServlet extends HttpServlet {
   protected void     doPost(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
      String   name = request.getParameter("name");
      String   comment = request.getParameter("comment");
      // Sanitize input to prevent XSS attacks
      name = sanitizeInput(name);
      comment = sanitizeInput(comment);
      // Process the sanitized data
      response.setContentType("text/html");
      Print Writer out = response.getWriter();
      out.print ln("<html><body>");
      out.print ln("<h2>Comment Received</h2>");
```

```
      out.print ln("<p>From: " + name + "</p>");
      out.print ln("<p>Comment: " + comment + "</p>");
      out.print ln("</body></html>");
   }
   private String  sanitizeInput(String  input) {
      if (input == null) {
         return "";
      }
      // Replace potentially dangerous char   acters with their HTML
```
entities
```
      String   sanitized = input
         .replace("&", "&amp;")
         .replace("<", "&lt;")
         .replace(">", "&gt;")
         .replace("\"", "&quot;")
         .replace("'", "&#x27;")
         .replace("/", "&#x2F;");
      return sanitized;
   }
}
```

CSRF Protection

Implement Cross-Site Request Forgery (CSRF) protection by using tokens:

```
@WebServlet("/secureForm")
public      class  SecureFormServlet extends HttpServlet {
   protected void    doGet(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
      // Generate a CSRF token
      String   csrfToken = generateCSRFToken();
      // Store the token in the session
      HttpSession session = request.getSession();
      session.setAttribute("csrfToken", csrfToken);
      response.setContentType("text/html");
      Print Writer out = response.getWriter();
      out.print ln("<html><body>");
      out.print ln("<h2>Secure Form</h2>");
```

```
        out.print ln("<form action='processSecureForm'
method='post'>");
        out.print ln("Name: <input type='text' name='name'><br>");
        out.print ln("Email: <input type='email' name='email'><br>");
        // Include the CSRF token as a hidden field
        out.print ln("<input type='hidden' name='csrfToken' value='" +
csrfToken + "'>");
        out.print ln("<input type='submit' value='Submit'>");
        out.print ln("</form>");
        out.print ln("</body></html>");
    }


    private String   generateCSRFToken() {
        // Generate a random token (in a real application, use a
cryptographically secure method)
        return UUID.randomUUID().toString  ();
    }
}

@WebServlet("/processSecureForm")
public      class  SecureFormProcessorServlet extends HttpServlet {
    protected void    doPost(HttpServletRequest request,
HttpServletResponse response)
          throws ServletException, IOException {
        // Retrieve the submitted token
        String   submittedToken = request.getParameter("csrfToken");
        // Retrieve the stored token from the session
        HttpSession session = request.getSession();
        String   storedToken = (String  )
session.getAttribute("csrfToken");
        response.setContentType("text/html");
        Print Writer out = response.getWriter();
        // Validate the CSRF token
        if (storedToken != null &&
storedToken.equals(submittedToken)) {
            // Token is valid, process the form
            String   name = request.getParameter("name");
            String   email = request.getParameter("email");
```

```
        out.print ln("<html><body>");
        out.print ln("<h2>Form Processed Successfully</h2>");
        out.print ln("<p>Name: " + name + "</p>");
        out.print ln("<p>Email: " + email + "</p>");
        out.print ln("</body></html>");
        // Invalidate the token after use (one-time use)
        session.removeAttribute("csrfToken");
    } else {
        // Invalid or missing token, potential CSRF attack
        response.setStatus(HttpServletResponse.SC_FORBIDDEN);
        out.print ln("<html><body>");
        out.print ln("<h2>Error: Invalid Request</h2>");
        out.print ln("<p>The form submission could not be processed
due to security concerns.</p>");
        out.print ln("</body></html>");
    }
  }
}
```

3.5 Handling Client Request and Generating Server Response

Java servlets operate on the fundamental principle of handling client requests and providing responses. This section walks through all aspects of this request-response cycle, from understanding what an HTTP protocol is to generating dynamic content based on user input..

(vii) Understanding the HTTP Request-Response Cycle

In order to understand the specifics of how to handle requests in servlets, we need to learn the HTTP request-response cycle::

Client Request: The client (typically a web browser) sends an HTTP request to the server.

Server Processing: The server processes the request, which may involve:

Routing the request to the appropriate servlet

Extracting request parameters

Processing business logic

Accessing databases or external services

Server Response: The server generates an HTTP response and sends it back to the client.

Client Rendering: The client processes the response (e.g., rendering HTML, executing JavaScript).

In Java servlets, this cycle is represented by:

The HttpServletRequest object, which encapsulates the client request

The HttpServletResponse object, which provides methods to generate the response

(viii) Analyzing the Request

In order to handle an incoming request we need to understand it. Servlets offer several ways to get information from the request.

Request Headers

HTTP headers are metadata about the request. Using the getHeader() method you can fetch headers equal to:

```java
@WebServlet("/requestInfo")
public     class RequestInfoServlet extends HttpServlet {
  protected void              doGet(HttpServletRequest request, HttpServletResponse response)
       throws ServletException, IOException {
    response.setContentType("text/html");
    Print Writer out = response.getWriter();
    out.print ln("<html><body>");
    out.print ln("<h2>Request Information</h2>");
    // Get basic request information
    out.print ln("<h3>Basic Info</h3>");
    out.print ln("<p>Request Method: " + request.getMethod() + "</p>");
    out.print ln("<p>Request URI: " + request.getRequestURI() + "</p>");
    out.print ln("<p>Protocol: " + request.getProtocol() + "</p>");
    // Get request headers
    out.print ln("<h3>Request Headers</h3>");
    out.print ln("<table border='1'>");
    out.print     ln("<tr><th>Header     Name</th><th>Header Value</th></tr>");

    Enumeration<String          >     headerNames     = request.getHeaderNames();
    while (headerNames.hasMoreElements()) {
```

```
        String   headerName = headerNames.nextElement();
        String   headerValue = request.getHeader(headerName);
        out.print  ln("<tr><td>" +  headerName  +  "</td><td>" +
headerValue + "</td></tr>");
    }
    out.print ln("</table>");
    out.print ln("</body></html>");
  }
}
```

Cookie Information

Cookies sent by the client can be retrieved using the getCookies() method:

```
@WebServlet("/cookieInfo")
public       class  CookieInfoServlet extends HttpServlet {
  protected   void                 doGet(HttpServletRequest  request,
HttpServletResponse response)
        throws ServletException, IOException {
    response.setContentType("text/html");
    Print Writer out = response.getWriter();
    out.print ln("<html><body>");
    out.print ln("<h2>Cookie Information</h2>");
    Cookie[] cookies = request.getCookies();
    if (cookies != null && cookies.length > 0) {
       out.print ln("<table border='1'>");
       out.print      ln("<tr><th>Cookie       Name</th><th>Cookie
Value</th></tr>");
       for (Cookie cookie : cookies) {
          out.print ln("<tr>");
          out.print ln("<td>" + cookie.getName() + "</td>");
          out.print ln("<td>" + cookie.getValue() + "</td>");
          out.print ln("</tr>");
       }

       out.print ln("</table>");
    } else {
       out.print ln("<p>No cookies found in this request.</p>");
    }
```

```
      out.print ln("</body></html>");
   }
}
```

Session Information

HTTP sessions allow you to track user state across multiple requests:

```
@WebServlet("/sessionInfo")
public     class  SessionInfoServlet extends HttpServlet {
  protected   void                doGet(HttpServletRequest  request,
HttpServletResponse response)
       throws ServletException, IOException {
    response.setContentType("text/html");
    Print Writer out = response.getWriter();
    // Get or create a session
    HttpSession session = request.getSession();

    // Update session access counter
    Int      eger     accessCount    =    (Int      eger)
session.getAttribute("accessCount");
    if (accessCount == null) {
      accessCount = 1;
    } else {
      accessCount++;
    }
    session.setAttribute("accessCount", accessCount);
    out.print ln("<html><body>");
    out.print ln("<h2>Session Information</h2>");
    out.print ln("<p>Session ID: " + session.getId() + "</p>");
    out.print  ln("<p>Session  Creation  Time:  "  +  new
Date(session.getCreationTime()) + "</p>");
    out.print  ln("<p>Last  Accessed  Time:  "  +  new
Date(session.getLastAccessedTime()) + "</p>");
    out.print ln("<p>Is New     Session: " + session.isNew     () +
"</p>");
    out.print  ln("<p>Session  Access  Count:  "  +  accessCount  +
"</p>");

    out.print ln("</body></html>");
  }
```

}

Request Attributes

Servlets can set and retrieve attributes within each request scope, which is useful for storing information relevant to those components.:

```java
@WebServlet("/setAttributes")
public class AttributeSetterServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        // Set some request attributes
        request.setAttribute("username", "john_doe");
        request.setAttribute("userRole", "admin");
        request.setAttribute("lastLogin", new Date());
        // Forward the request to another servlet to display the attributes
        RequestDispatcher dispatcher = request.getRequestDispatcher("/displayAttributes");
        dispatcher.forward(request, response);
    }
}

@WebServlet("/displayAttributes")
public class AttributeDisplayServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html");
        Print Writer out = response.getWriter();
        out.print ln("<html><body>");
        out.print ln("<h2>Request Attributes</h2>");
        // Retrieve and display attributes
        String username = (String ) request.getAttribute("username");
        String userRole = (String ) request.getAttribute("userRole");
        Date lastLogin = (Date) request.getAttribute("lastLogin");
        out.print ln("<p>Username: " + username + "</p>");
        out.print ln("<p>User Role: " + userRole + "</p>");
        out.print ln("<p>Last Login: " + lastLogin + "</p>");
        out.print ln("</body></html>");
    }
}
```

(ix) Generating the Response

Now, servlets must provide a proper reply after handling the request. You can create different types of responses using HttpServletResponse object.

Setting Response Headers

Response headers provide metadata about the response:

```
@WebServlet("/setHeaders")
public      class  HeaderSetterServlet extends HttpServlet {
   protected void    doGet(HttpServletRequest request,
HttpServletResponse response)
         throws ServletException, IOException {
     // Set response headers
     response.setContentType("text/html");
     response.setHeader("Cache-Control", "no-cache, no-store, must-
revalidate");
     response.setHeader("Pragma", "no-cache");
     response.setHeader("Expires", "0");
     response.setHeader("Custom-Header", "Custom Value");
     Print Writer out = response.getWriter();
     out.print ln("<html><body>");
     out.print ln("<h2>Custom Headers Set</h2>");
     out.print ln("<p>This response includes custom HTTP headers
that control caching and demonstrate header setting.</p>");
     out.print ln("</body></html>");
   }
}
```

Setting Cookies

Cookies allow you to store small pieces of data on the client:

```
@WebServlet("/setCookie")
public      class  CookieSetterServlet extends HttpServlet {
   protected void    doGet(HttpServletRequest request,
HttpServletResponse response)
         throws ServletException, IOException {
     // Create a new      cookie
     Cookie userCookie = new      Cookie("username", "john_doe");
     // Configure the cookie
     userCookie.setMaxAge(24 * 60 * 60); // Expires in 24 hours
```

```
    userCookie.setPath("/");        // Available across the entire
application
    userCookie.setHttpOnly(true);      // Not accessible via
JavaScript
    userCookie.setSecure(true);       // Only sent over HTTPS
    // Add the cookie to the response
    response.addCookie(userCookie);
    // Create a session tracking cookie
    Cookie trackingCookie = new      Cookie("sessionTracker",
UUID.randomUUID().toString ());
    trackingCookie.setMaxAge(30 * 60);  // Expires in 30 minutes
    response.addCookie(trackingCookie);
    response.setContentType("text/html");
    Print Writer out = response.getWriter();
    out.print ln("<html><body>");
    out.print ln("<h2>Cookies Set</h2>");
    out.print ln("<p>The following cookies have been set:</p>");
    out.print ln("<ul>");
    out.print ln("<li>username: john_doe (expires in 24
hours)</li>");
    out.print ln("<li>sessionTracker: " + trackingCookie.getValue() +
" (expires in 30 minutes)</li>");
    out.print ln("</ul>");
    out.print ln("</body></html>");
  }
}
HTTP Status Codes
Setting the appropriate HTTP status code is important for proper
client-server communication:
@WebServlet("/statusCodes")
public     class StatusCodeDemoServlet extends HttpServlet {
  protected void    doGet(HttpServletRequest request,
HttpServletResponse response)
      throws ServletException, IOException {
    String  codeParam = request.getParameter("code");
    if (codeParam != null) {
      try {
        int  statusCode = Int eger.parseInt (codeParam);
```

```
switch (statusCode) {
    case 200:
        response.setStatus(HttpServletResponse.SC_OK);
        sendResponse(response, "200 OK", "The request has
succeeded.");
        break;
    case 201:
response.setStatus(HttpServletResponse.SC_CREATED);
    sendResponse(response, "201 Created", "The request has been
fulfilled and a new      .
```

(ix) Handling Cookies

Cookies are one of the core technologies that allow managing state on web apps. One of the challenges developers who use HTTP protocol face is that it is stateless. Cookies became the elegant solution to this problem, peas in a pod of data that could be stored on the client's side and sent with every request. Specifically in Java web development, within the Servlets and JSP framework, cookies miss an elegant way to persist user settings, track user's activities, and maint ain the state of a user's session. In a way, a cookie is just a small text file, which is stored in the client browser. When a user visits a site, the server can send one or more cookies to that user's browser, which the browser keeps locally. When using the same server on additional requests, the browser automatically adds these cookies to the request headers. This mechanism enables the server to identify returning users and pull up previously stored information without relying on users needing to int roduce themselves on each and every page request. The Java Servlet API provides a rich set of class es and methods that can be used to create, modify, and retrieve cookies. The main class for cookie operations is javax. servlet. http. Cookie: This is just a handy way to encapsulate the name/value pairs that make up a cookie. This API allows the Java developer to work with cookies in their web applications, providing a rich, personalized experience for the user.

Cookies offer a key feature in modern web development, allowing websites to remember user preferences, store their shopping cart data, implement authentication mechanisms, and facilitate personalized content delivery. But in recent years with GDPR, the CCPA and growing focus on user privacy, the standard use of cookies by

developers means they have to be careful about how they implement cookie-based solutions. We will cover the technical details related to cookies in Java web applications but also some important concerns around privacy, security, and best practices.

This article will cover deeper cookie management — the attributes used to specify cookie behaviour, how cookies are sent and received, removing cookies and the benefits and drawbacks of using cookies. We will also delve int o how cookies fit int o larger session management paradigms, discussing the use of cookies, session tracking mechanisms, and their achievements to provide all-encompassing state preservation in Java web applications.

(x) Handling Cookies in Java Web Applications: The javaxz.servlet.http package in Java's Servlet API enables powerful cookie support. servlet. http. Cookie class . In this section you study core functionality for creating, sending and receiving, and manipulating cookies in Java web applications.

(xi) Creating and Sending Cookies: In Java, a cookie can be created easily. The Cookie class 's constructor takes the cookie name and cookieValue as String parameters. You create a cookie, and then send it to the client browser through the response. addCookie() method. Here is the process translated int o an example::

// Create a new cookie
Cookie userCookie = new Cookie("username", "john_doe");
// Set cookie properties (optional)
userCookie.setMaxAge(60 * 60 * 24 * 30); // Expires in 30 days (in seconds)
userCookie.setPath("/");   // Available across the entire application
userCookie.setSecure(true);   // Only sent over HTTPS
userCookie.setHttpOnly(true);   // Not accessible via JavaScript
// Send the cookie to the client
response.addCookie(userCookie);

Here, we have created a cookie with a name "username" and value "john_doe". Then, we set multiple properties and set it to send it to client browser. These attributes determine the cookie's behavior, such as its duration, accessibility, and security properties

(xii) Receiving and Reading Cookies

All cookies for the domain are included in request headers when a client does a request to the server. You retrieve these cookies in a servlet, by using the request. getCookies() method which return array of Cookie objects. The code below shows how to obtain and read the cookies:

```
// Get all cookies from the request
Cookie[] cookies = request.getCookies();
// Check if cookies exist
if (cookies != null) {
    // Iterate through all cookies
    for (Cookie cookie : cookies) {
        // Retrieve the cookie name and value
        String   name = cookie.getName();
        String   value = cookie.getValue();
        // Process the cookie based on its name
        if ("username".equals(name)) {
            // Found the username cookie
            System.out.print ln("Welcome back, " + value);
            break;
        }
    }
}
```

The above code iterates through all cookies received in the request, searching for a specific cookie by name. Once found, the cookie's value can be retrieved and used to customize the response or make application decisions.

(xiii) Modifying Cookies

The above code loops through all the cookies that were sent with the request and looks for one with a specific name. When located, the cookie value can be accessed, and the data can be used to tailor the response or to decide on actions to take within the application:

```
// Get all cookies from the request
Cookie[] cookies = request.getCookies();

if (cookies != null) {
    for (Cookie cookie : cookies) {
```

```
if ("username".equals(cookie.getName())) {
    // Create a new    cookie with the same name but updated value
    Cookie  updatedCookie  =  new           Cookie("username",
"jane_doe");
    updatedCookie.setMaxAge(cookie.getMaxAge());
    updatedCookie.setPath(cookie.getPath());

    // Send the updated cookie to the client
    response.addCookie(updatedCookie);
    break;
    }
  }
}
```

In this example, we search for the "username" cookie and create a new cookie with the same name but an updated value. We also preserve the original cookie's attributes to ensure consistent behavior.

(xiv) Deleting Cookies

In order to remove a cookie, set its age to zero or a negative value and send it back to the client. This is an instruction in your web browser to delete the cookie. Here is some code that shows how this can be done:

```
// Create a cookie with the same name
Cookie cookieToDelete = new      Cookie("username", "");
// Set the maximum age to 0 (delete immediately)
cookieToDelete.setMaxAge(0);
// Ensure it's on the same path as the original cookie
cookieToDelete.setPath("/");
// Send the cookie to the client
response.addCookie(cookieToDelete);
```

You must set the path for the cookie being deleted to be the same as the original cookie. If the paths differ, the browser may not treat it as the same cookie, so the deletion will silently fail

Cookie Persistence

Cookies can be class ified int o two types based on their persistence:

(xv) Session Cookies: These cookies expire when the browser session ends. They are stored in memory and are not written to disk. To create a session cookie, don't set the maxAge property or set it to -

```
Cookie sessionCookie = new        Cookie("sessionId",
generateSessionId());
// No maxAge means it's a session cookie
response.addCookie(sessionCookie);
```

(xvi) Persistent Cookies: These cookies have a specific expiration time and are stored on disk. They persist even after the browser is closed and are sent with requests until they expire.

```
Cookie persistentCookie = new        Cookie("preferredLanguage",
"en");
persistentCookie.setMaxAge(60 * 60 * 24 * 365); // 1 year in seconds
response.addCookie(persistentCookie);
```

Choosing between session and persistent cookies depends on the application's requirements and the nature of the data being stored.

(xvii) Benefits of Using Cookies

Cookies offer numerous advantages for web applications, particularly in the context of Java-based systems. This section explores the key benefits of incorporating cookies int o your application architecture.

(xviii) User Experience Enhancement: One of the primary benefits of cookies is their ability to enhance user experience by remembering user preferences and settings. Consider a web application that allows users to customize the int erface, such as choosing a theme or language. By storing these preferences in cookies, the application can provide a consistent experience across multiple visits without requiring users to reconfigure their settings each time.

```
// Example: Storing user theme preference
String  selectedTheme = request.getParameter("theme");
if (selectedTheme != null && !selectedTheme.isEmpty()) {
   Cookie  themeCookie  =  new              Cookie("userTheme",
selectedTheme);
   themeCookie.setMaxAge(60 * 60 * 24 * 365); // 1 year
   themeCookie.setPath("/");
   response.addCookie(themeCookie);
}
```

This kind of personalization significantly improves user satisfaction and engagement by creating a tailored experience that acknowledges and respects individual preferences.

(xix) State Management in Stateless Protocols: HTTP is stateless by design, that is, every request to the server is considered independent and does not know about prior requests. Cookies allow you to maintain state across multiple requests. For example, cookie functions in insurance apps for shopping carts are track selected items:

```
// Example: Adding item to cart (simplified)
String itemId = request.getParameter("itemId");
if (itemId != null) {
    // Get existing cart cookie
    String cartItems = "";
    Cookie[] cookies = request.getCookies();
    if (cookies != null) {
        for (Cookie cookie : cookies) {
            if ("cartItems".equals(cookie.getName())) {
                cartItems = cookie.getValue();
                break;
            }
        }
    }
    // Add new      item to cart
    if (!cartItems.isEmpty()) {
        cartItems += "," + itemId;
    } else {
        cartItems = itemId;
    }
    // Update cart cookie
    Cookie cartCookie = new      Cookie("cartItems", cartItems);
    cartCookie.setMaxAge(60 * 60 * 24 * 7); // 1 week
    cartCookie.setPath("/");
    response.addCookie(cartCookie);
}
```

Whenever you visit a store and start browsing, you can add things to your cart, and it goes around without you losing what you've selected for a smooth shopping experience.

(xx) Performance Optimization: When used correctly, cookies can greatly enhance the performance of the application by avoid ing database queries or server-side storage. You are also enabled for the terrible site fetches if stored in cookies for non-sensitive, frequently accessed data, which can reduce server load and improve response times. For instance, placing display preferences or non-sensitive user data int o cookies can save you from needing to pull this data from the database on each request:

```
// First-time user setup
if (request.getCookies() == null ||
!containsCookie(request.getCookies(), "displaySettings")) {
    // Default settings
    Cookie settingsCookie = new Cookie("displaySettings",
"compact:true,showImages:true,fontSize:medium");
    settingsCookie.setMaxAge(60 * 60 * 24 * 30); // 30 days
    settingsCookie.setPath("/");
    response.addCookie(settingsCookie);
}

// Helper method to check if a cookie exists
private boolean containsCookie(Cookie[] cookies, String name) {
    for (Cookie cookie : cookies) {
        if (name.equals(cookie.getName())) {
            return true;
        }
    }
    return false;
}
```

Client-side storage also helps offload data to the front-end which ultimately relieves the database service and results in quicker response times and improved scalability.

(xxi) Authentication and Remember Me Functionality

Cookies are essential for implementing "Remember Me" functionality, which allows users to remain authenticated across browser sessions without re-entering credentials. This feature significantly enhances user convenience while maint aining security.

// Example: Implementing "Remember Me" functionality

```
boolean rememberMe=
"true".equals(request.getParameter("rememberMe"));
if (rememberMe) {
    // Generate secure token (simplistic example)
    String   rememberToken = generateSecureToken(username);
    // Store token in database (associated with user)
    storeRememberTokenInDatabase(username, rememberToken);
    // Create persistent cookie with token
    Cookie  rememberCookie  =  new          Cookie("rememberToken",
rememberToken);
    rememberCookie.setMaxAge(60 * 60 * 24 * 30); // 30 days
    rememberCookie.setHttpOnly(true); // Prevent JavaScript access
    rememberCookie.setSecure(true);   // HTTPS only
    rememberCookie.setPath("/");
    response.addCookie(rememberCookie);
}
```

In this example, a secure token is generated, stored in the database, and also sent to the client as a cookie. On subsequent visits, the application can validate this token to automatically authenticate the user without requiring a new   login.

(xxii) Analytics and User Behavior Tracking

Cookies are useful for tracking user behavior and collecting analytics data. This enables applications to track navigation patterns, feature usage, and user preferences by assigning unique identifiers to visitors.

```
// Example: Setting analytics tracking cookie
String   visitorId = UUID.randomUUID().toString  ();
Cookie analyticsCookie = new       Cookie("visitorId", visitorId);
analyticsCookie.setMaxAge(60 * 60 * 24 * 365 * 2); // 2 years
analyticsCookie.setPath("/");
response.addCookie(analyticsCookie);
// Log page visit
logPageVisit(visitorId, request.getRequestURI());
```

This helps product development, marketing strategies and int erface refinements, which in turn contribute to improved user experiences and business results.

(xxiii) Cookie Attributes and Security Considerations

In addition to the simple name-value pair, cookies also support a range of attributes which can influence their behavior, scope, and security char acteristics. "It is important to comprehend these attributes if you want to deploy safe and efficient cookie-based solutions..

Domain and Path Attributes: Domain and Path attributes help us identify the URLs to which a cookie needs to be sent..

Domain — The dot char acter (.) specifies the domain for which the cookie is valid. A cookie is, by default, sent only to the domain that set it. But you can set a cookie accessible to subdomains by providing a domain prepended with a dot.

Cookie domainCookie = new Cookie("sitePreferences", "darkMode:true");

domainCookie. setDomain(". example. com"); // Only available on example. com

response. addCookie(domainCookie);.

Path Attribute: Specifies the portion of the URL path that must exist in the requested resource before sending the Cookie header. Cookies are by default set for the path of the URL where the setting occurs. Domain and Path Example: Setting the path to "/" will make the cookie accessible across the entire domain.

Cookie pathCookie = new Cookie("shopCart", "item1:3,item2:1");
pathCookie.setPath("/shop"); // Only available to URLs starting with /shop

response.addCookie(pathCookie);

In this example, the cookie will be sent only to pages under the /shop path, such as /shop/cart and /shop/products.

Secure and HttpOnly Flags

These flags enhance cookie security by restricting when and how cookies are transmitted and accessed.

Secure Flag: When set, the cookie is only sent over HTTPS connections, protecting it from int erception over unsecured channels.

Cookie secureCookie = new Cookie("authToken", generateToken());

secureCookie.setSecure(true); // Only sent over HTTPS

response.addCookie(secureCookie);

This is particularly important for cookies containing sensitive information like authentication tokens.

(xxiv) HttpOnly Flag: Prevents client-side JavaScript from accessing the cookie, mitigating the risk of cross-site scripting (XSS) attacks.
Cookie httpOnlyCookie = new   Cookie("sessionId", sessionId);
httpOnlyCookie.setHttpOnly(true); // Not accessible via JavaScript
response.addCookie(httpOnlyCookie);
By using the HttpOnly flag, even if an attacker manages to inject malicious JavaScript int o your page, they won't be able to access the cookie directly.

(xxv) SameSite Attribute (Servlet API 4.0+): The SameSite attribute, int roduced in new   er servlet specifications, controls whether cookies are sent with cross-site requests, providing protection against cross-site request forgery (CSRF) attacks.
Cookie sameSiteCookie = new   Cookie("csrfToken", generateCSRFToken());
sameSiteCookie.setAttribute("SameSite", "Strict"); // Only sent in same-site context
response.addCookie(sameSiteCookie);
The SameSite attribute can have three values:
Strict: The cookie is only sent in a first-party context.
Lax: The cookie is sent with top-level navigations and with GET requests from other sites.
None: The cookie is sent in all contexts, including cross-site requests. Note that when using SameSite=None, the cookie must also have the Secure flag set.
7.4.4 Expiration and MaxAge
The expiration time of a cookie can be controlled using the setMaxAge() method, which specifies the cookie's lifespan in seconds.
// Session cookie (expires when the browser is closed)
Cookie sessionCookie = new   Cookie("tempData", "value");
sessionCookie.setMaxAge(-1); // Default behavior for session cookies
response.addCookie(sessionCookie);
// Persistent cookie (expires after a specific time)
Cookie persistentCookie = new   Cookie("userPrefs", "theme:dark");
persistentCookie.setMaxAge(60 * 60 * 24 * 30); // 30 days in seconds
response.addCookie(persistentCookie);

// Delete a cookie

Cookie deleteCookie = new   Cookie("oldCookie", "");

deleteCookie.setMaxAge(0); // Expires immediately

response.addCookie(deleteCookie);

The MaxAge value determines whether a cookie is stored temporarily in memory or persistently on disk, and for how long it remains valid.

(xxv) Cookie Size Limitations: Browsers impose limits on cookie size and the number of cookies allowed per domain. These limitations vary by browser but generally include:

- Maximum size per cookie: Usually around 4KB
- Maximum number of cookies per domain: Typically 50-60
- Maximum total size of all cookies for a domain: Around 4KB to 10KB

Given these constraint s, it's important to use cookies efficiently:

// BAD PRACTICE: Storing large data in cookies

Cookie largeCookie = new   Cookie("userData", largeJsonObject);

// May exceed limits

// BETTER PRACTICE: Store minimal data in cookies

Cookie idCookie = new   Cookie("userId", "12345");

response.addCookie(idCookie);

// Retrieve additional data from server-side storage as needed

For large amounts of data, consider alternatives like HTML5 Web Storage (localStorage/sessionStorage) or IndexedDB, with cookies used primarily for authentication and session management.

Cookie Security Best Practices :Implementing secure cookie practices is essential for protecting user data and preventing common attacks:

 Use the Secure flag for sensitive cookies**:**

authCookie.setSecure(true);

Apply the HttpOnly flag to prevent XSS attacks:

authCookie.setHttpOnly(true);

Implement proper cookie expiration**:**

// Set reasonable expiration times based on the cookie's purpose

authCookie.setMaxAge(60 * 30); // 30 minutes for authentication

Validate cookie data**:**

String   cookieValue = cookie.getValue();

if (isValidFormat(cookieValue)) {

```
   // Process the cookie
} else {
   // Handle invalid data (potential tampering)
}
```

 Encrypt sensitive cookie values**:**

```
// Example of encrypting cookie value
String   encryptedValue = encryptData(rawValue, encryptionKey);
Cookie encryptedCookie = new      Cookie("sensitiveData",
encryptedValue);
```

Implement CSRF protection alongside cookies**:**

```
// Generate and store CSRF token
String   csrfToken = generateRandomToken();
Cookie csrfCookie = new      Cookie("csrfToken", csrfToken);
csrfCookie.setHttpOnly(false); // Allow JavaScript access for form
submission
response.addCookie(csrfCookie);
// Store token in session for server-side verification
session.setAttribute("csrfToken", csrfToken);
```

By following these best practices, developers can leverage the benefits of cookies while minimizing security risks.

12.10: Session Tracking in Java Web Applications

In contrast, cookies are a mechanism for storing small bits of information on the client side, but come with limitations in terms of size, count, and security. Whereas cookie is limited to a single request, session tracking is used to maint ain status between multiple requests.

(i) Need for Session Tracking: The statelessness of HTTP poses great difficulties for int eractive web application development. You have no context beyond the input you received with every request. This limitation presents a problem in situations like:

- **Multi-step processes:** These are operations such as checkout workflows, multi-page forms, or wizard int erfaces that involve multiple steps and require maint aining state across multiple requests.
- **User authentication:** Remembering who is logged-in without asking for credentials on every request.

- **Application state:** Value can be used to to keep and manage complex state for an application, such as shopping carts, game states, or workspace configurations.
- **Customization:** Providing tailored content based on user preference or browsing history.

(ii) Session Tracking

The Servlet specification in Java has support for multiple session tracking mechanisms:

- **Cookie-Based Sessions :** The server creates it and sends it to the client as a cookie. This cookie is included in subsequent requests, permitting the server to identify the session.
- **URL Rewriting:** For those browsers that do not support or have disabled cookies, at the end of the URLs the session ID may be appended as a parameter.
- **SSL Sessions:** The SSL session ID can be used to keep the session state for HTTPS connections without cookies or URL parameters.

Hidden Form Fields One way is to use session IDs as hidden fields in HTML forms and post them along with form data. Out of which, session through cookies is the most common and reliable way to implement it, whereas URL rewriting could be fall back when no cookies available Of these mechanisms, cookie-based sessions are the most common and reliable approach, with URL rewriting often used as a fallback when cookies are unavailable.

(iii) The HttpSession API: Java's Servlet API offers complete int eraction with session management using the HttpSession int erface. This means developers can use this API for session tracking without worrying about the underlying mechanism.

(iv) Creating or Retrieving a Session:

// Get the current session, or create one if it doesn't exist

HttpSession session = request.getSession();

// Get the current session only if it exists, without creating a new one

HttpSession existingSession = request.getSession(false);

The request.getSession() method returns the current session object associated with the request. If no session exists, it creates a new

one automatically. This behavior can be controlled using the boolean parameter: request.getSession(boolean create).

(v) Storing and Retrieving Data in Sessions:

```
// Store data in the session
session.setAttribute("username", "john_doe");
session.setAttribute("logint ime", new      Date());
session.setAttribute("shoppingCart", cartObject);
// Retrieve data from the session
String   username = (String  ) session.getAttribute("username");
Date logint ime = (Date) session.getAttribute("logint ime");
ShoppingCart cart = (ShoppingCart)
session.getAttribute("shoppingCart");
// Remove data from the session
session.removeAttribute("temporaryData");
```

The session acts as a map-like structure, storing attributes as key-value pairs. These attributes can be of any Java type, including complex objects, as long as they implement the Serializable int erface.

(vi) Managing Session Lifecycle:

```
// Get session creation time
long creationTime = session.getCreationTime();
// Get last accessed time
long lastAccessTime = session.getLastAccessedTime();
// Set session timeout (in seconds)
session.setMaxInactiveInt erval(1800); // 30 minutes
// Invalidate (terminate) the session
session.invalidate();
```

The session timeout specifies how long the session remains active without client int eraction. After the specified period of inactivity, the server automatically invalidates the session. Sessions can also be explicitly invalidated using the invalidate() method, typically during logout operations.

(vii) Accessing Session Metadata:

```
// Get the session ID
String   sessionId = session.getId();

// Check if this is a new      session
```

boolean isNew   = session.isNew   ();

// Get the maximum inactive int erval
int maxInactiveInt erval = session.getMaxInactiveInt erval();
These methods provide access to session metadata, which can be
useful for debugging, logging, and session management operations.

(viii) Session Tracking Implementation Examples
Let's explore some practical examples of session tracking in Java web
applications:
Example 1: User Authentication and Authorization
@WebServlet("/login")
public   class LoginServlet extends HttpServlet {
  @Override
  protected void   doPost(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
    String   username = request.getParameter("username");
    String   password = request.getParameter("password");
    // Validate credentials (simplified example)
    if (isValidUser(username, password)) {
      // Get the session (create if it doesn't exist)
      HttpSession session = request.getSession();
      // Store user information in the session
      User user = getUserDetails(username);
      session.setAttribute("user", user);
      session.setAttribute("authenticated", true);
      session.setAttribute("logint ime", new   Date());
      // Set session timeout (30 minutes)
      session.setMaxInactiveInt erval(30 * 60);
      // Redirect to dashboard
      response.sendRedirect("dashboard");
    } else {
      // Authentication failed - redirect back to login page
      request.setAttribute("errorMessage", "Invalid username or
password");
        request.getRequestDispatcher("/login.jsp").forward(request,
response);

```
    }
  }
  // Validation methods (implementation details omitted)
  private boolean isValidUser(String   username, String   password) {
/* ... */ }
  private User getUserDetails(String   username) { /* ... */ }
}
```

This example demonstrates how sessions can be used to track authenticated users. After successful authentication, user information is stored in the session, allowing subsequent requests to verify the user's identity without re-authenticating.

Example 2: Shopping Cart Implementation

```
@WebServlet("/cart/*")
public      class  ShoppingCartServlet extends HttpServlet {
  @Override
  protected void     doGet(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
    // Get the current session (don't create a new      one)
    HttpSession session = request.getSession(false);
    if (session == null) {
      // No session exists - redirect to homepage
      response.sendRedirect("/home");
      return;
    }
    // Retrieve cart from session
    ShoppingCart cart = (ShoppingCart) session.getAttribute("cart");
    if (cart == null) {
      // Initialize cart if it doesn't exist
      cart = new      ShoppingCart();
      session.setAttribute("cart", cart);
    }

    // Display cart contents
    request.setAttribute("cartItems", cart.getItems());
    request.setAttribute("totalPrice", cart.getTotalPrice());
    request.getRequestDispatcher("/cart.jsp").forward(request,
response);
```

```
        }

        @Override
        protected void    doPost(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        // Get current session or create one
        HttpSession session = request.getSession();
        // Get cart from session or create a new      one
        ShoppingCart cart = (ShoppingCart) session.getAttribute("cart");
        if (cart == null) {
            cart = new      ShoppingCart();
            session.setAttribute("cart", cart);
        }
        // Process cart operation
        String   action = request.getParameter("action");
        if ("add".equals(action)) {
            // Add item to cart
            String   productId = request.getParameter("productId");
            int  quantity = Int eger.parseInt
(request.getParameter("quantity"));
            cart.addItem(productId, quantity);
        } else if ("remove".equals(action)) {
            // Remove item from cart
            String   productId = request.getParameter("productId");
            cart.removeItem(productId);
        } else if ("clear".equals(action)) {
            // Clear cart
            cart.clear();
        }
        // Redirect back to cart display
        response.sendRedirect("/cart");
    }
}
```

In this example, we highlight a cart functionality implemented to
keep track of the items in your session. This cart object is saved in the
session object giving the user the ability to add, delete, and view items
while making multiple requests..

Session Management Best Practices

Session management needs to be done with the utmost attention to detail with regards to security, performance, and user experience:

(ix) Security Considerations:

Session ID Protection:

```java
// Configure the session cookie to be secure and HttpOnly
@WebServlet("/secureApp")
public class SecureAppServlet extends HttpServlet {
    @Override
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        // Configure session cookies
        ServletContext context = config.getServletContext();
        context.getSessionCookieConfig().setHttpOnly(true);
        context.getSessionCookieConfig().setSecure(true);
    }
    // Servlet methods...
}
```

Session Fixation Prevention:

```java
// After successful authentication, regenerate the session ID
@WebServlet("/login")
public class SecureLoginServlet extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        // Authenticate user...
        // After successful authentication
        if (authenticated) {
            // Get current session data
            HttpSession oldSession = request.getSession();
            Map<String, Object> attributes = new HashMap<>();
            Enumeration<String> names =
oldSession.getAttributeNames();
            while (names.hasMoreElements()) {
                String name = names.nextElement();
```

```java
                attributes.put(name, oldSession.getAttribute(name));
        }
        // Invalidate current session
        oldSession.invalidate();
        // Create new       session
        HttpSession new      Session = request.getSession(true);
        // Copy attributes to new        session
        for (Map.Entry<String  , Object> entry : attributes.entrySet())
{
            new       Session.setAttribute(entry.getKey(),
entry.getValue());
        }
        // Set authentication flag
        new      Session.setAttribute("authenticated", true);
    }
  }
}
```

Proper Session Termination**:**

```java
@WebServlet("/logout")
public        class  LogoutServlet extends HttpServlet {
  @Override
  protected void     doGet(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {

    // Get current session
    HttpSession session = request.getSession(false);

    if (session != null) {
      // Invalidate the session
      session.invalidate();
    }

    // Clear authentication cookie if used
    Cookie authCookie = new       Cookie("authToken", "");
    authCookie.setMaxAge(0);
    authCookie.setPath("/");
    response.addCookie(authCookie);
```

```
        // Redirect to login page
        response.sendRedirect("/login");
    }
}
```

(x) Performance Optimization:
Minimize Session Data: Store only necessary data in the session to reduce memory consumption and serialization/deserialization overhead.
Session Timeout Management: Balance security and user experience when setting session timeouts:
// Short timeout for sensitive operations
session.setMaxInactiveInt erval(900); // 15 minutes

// Longer timeout for regular browsing
session.setMaxInactiveInt erval(3600); // 1 hour
Session Clustering and Persistence**:** For high-availability applications, configure session replication or persistence:
<!-- Example Tomcat context.xml configuration -->
<Context>
<Manager class
Name="org.apache.catalina.session.PersistentManager"
        saveOnRestart="true">
<Store class Name="org.apache.catalina.session.FileStore"
        directory="/tmp/sessions"/>
</Manager>
</Context>

Check Your Progress

1. **Write a short servlet program to read and process form data using request.getParameter() method.**

   ----

2. **Explain the concept of session tracking in Java. Compare cookies, URL rewriting, and HttpSession-based approaches.**

12.11: Summary

This unit provides a detailed understanding of how servlets operate within web applications. The **Servlet Life Cycle** includes several stages — *loading, initialization (init()), request handling (service() / doGet() / doPost()), and destruction (destroy())* — managed automatically by the servlet container.

During **request processing**, the servlet receives an HttpServletRequest object containing client data and sends back an HttpServletResponse with the generated output. These int eractions typically occur via HTTP methods like GET and POST.

Session management ensures continuity of user int eractions across multiple requests. Java offers various mechanisms such as **cookies**, **URL rewriting**, **hidden form fields**, and **HttpSession objects** to maint ain session state.

Servlet security covers authentication (verifying user identity) and authorization (granting access). Security can be implemented declaratively using deployment descriptors (web.xml) or programmatically within the code. Techniques such as HTTPS and encryption ensure data privacy.

Int egration with **Java EE/Jakarta EE** enables servlets to work alongside technologies like JSP for dynamic content, JDBC for database connectivity, and EJB for business logic. Servlets can also handle **form data** using methods like request.getParameter() to process user input efficiently.

Lastly, **session tracking** techniques allow developers to design user-friendly, stateful web applications where data persists between multiple client-server exchanges, ensuring a smooth and personalized experience.

12.12: Exercises

Multiple Choice Questions (MCQs)

1. Which method is called only once when a servlet is first loaded int o memory?

a) service()

b) init()

c) destroy()

d) doPost()

Answer: b)

2. Which int erface must all servlets implement either directly or indirectly?

   a) HttpRequest

   b) ServletContext

   c) Servlet

   d) ServletResponse

   Answer: c)

3. Which object is used to track user sessions in servlets?

   a) HttpRequest

   b) HttpSession

   c) ServletConfig

   d) CookieManager

   Answer: b)

4. What is the main role of doPost() method in a servlet?

   a) To handle GET requests

   b) To handle POST requests from client forms

   c) To initialize servlet configuration

   d) To destroy servlet instance

   Answer: b)

5. Which of the following is NOT a session tracking technique in Java?

   a) Cookies

   b) Hidden Form Fields

   c) IP Address Matching

   d) URL Rewriting

   Answer: c)

Descriptive Questions

1. Explain the **servlet life cycle** and describe each of its stages with the respective methods.

2. Discuss the process of **HTTP request and response handling** in servlets with the help of doGet() and doPost() methods.

3. Describe the **different techniques of session management** in Java web applications and their advantages.

4. What are the **security mechanisms** available in servlets? Explain authentication and authorization in detail.

5. How can a servlet be int egrated with **Java EE/Jakarta EE** technologies like JSP and JDBC? Give suitable examples.

12.13: References and Suggested Readings

- Marty Hall & Larry Brown, *Core Servlets and JavaServer Pages*, Prentice Hall, 2023.

- Oracle, *Java Servlet Technology Documentation*, Oracle Corporation, 2024.

- Bryan Basham, Kathy Sierra & Bert Bates, *Head First Servlets and JSP*, O'Reilly Media, 2022.

- Cay S. Horstmann, *Core Java Volume II – Advanced Features*, Pearson Education, 2023.

- Jakarta EE Working Group, *Jakarta Servlet Specification 6.0*, Eclipse Foundation, 2024.

- Baeldung Tutorials, *Java Servlets and Session Management*, https://www.baeldung.com/

- Oracle Java EE Guide, *Session Tracking and Security in Servlets*, https://docs.oracle.com/javaee/

Block 4: JSP Technology

Unit 13: Introduction to JSP

Structure

13.1: Introduction

JavaServer Pages (JSP) is a server-side technology used to develop dynamic and platform-independent web applications. It simplifies the process of building web pages that display dynamically generated content, combining the power of Java with the flexibility of HTML. JSP is built on top of servlet technology, meaning every JSP page is int ernally converted int o a servlet by the web container before execution. A typical JSP page contains a mixture of HTML, JSP scripting elements, and Java code. JSP supports various elements such as declarations, scriptlets, expressions, and directives that help embed Java logic directly within web content. The JSP architecture separates presentation from business logic, enhancing maint ainability and reducing code duplication.When a user requests a JSP page, the container translates it int o a servlet, compiles it, and executes it to generate a dynamic response. This architecture ensures high performance and easy int egration with databases and backend systems. JSP supports Expression Language (EL) for simplified data access and Custom Tags for reusable components. With its clean separation of concerns, JSP remains a crucial part of enterprise Java development, forming the foundation of frameworks like JSF, Struts, and Spring MVC.

13.2: Learning Outcomes

1. To understand the concept, need, and benefits of JSP.
2. To explore the life cycle of JSP.

3. To study scripting elements and implicit objects.
4. To analyze directive elements and action elements in JSP.

13.3: JSP, Need and Benefit of JSP

JSP(JavaServer Pages) server-side technology to create dynamic web pages and web applications. Java Server Pages (JSP) is a web application technology that is used to create dynamic web content. JSP separates presentation logic (HTML, CSS) from business logic (Java code), making web applications easier to maint ain and scalable. JSP allows the development of web pages that are created dynamically, responding to user actions, form submission results, and return values from databases, instead of static web pages that are always the same when accessed. This is done by embedding Java code in special delimiters () in an HTML page. During the run time, JSPs are converted int o Servlets, which makes it highly performant and reliable. Due to this feature, JSP is widely used in enterprise-level web applications, online portals, and content management systems that need to process data in real-time.

The need for JSP to come int o existence came because static web technologies at that time like HTML and JavaScript couldn't generate content dynamically on the server side. The alternative is to use Servlets, but it can be tiresome and less maint ainable when writing HTML inside Java class es using Servlets. This problem is overcome by JSP in that it allows developers to code using Java code within an HTML file. Platform independent is one of the main advantages of JSP as it can be executed on any OS that supports Java. Then, JSP also provides automatic session management, which makes it easier to manage user sessions as compared to handling it manually. Plus, it works in harmony with JavaBeans, JDBC, and other Java technologies to facilitate database connections and data management. One of the other major benefits is tag libraries (JSTL), enabling the code to be reused and improves code modularity and maint ainability. These benefits make JSP a popular choice for developing enterprise applications, e-commerce simply by using, and int eractive web platforms.

13.4: Life Cycle of JSP

A JSP page has three main stages during its life cycle: compilation, execution, and request handling. The JSP engine first checks if the requested JSP page has already been compiled when a client sends a request for a JSP page. If not, it compiles the JSP file to a Servlet class . The translation step converts JSP constructs, such as scriptlets (), expressions (), and directives (), int o corresponding Java code. Once translated, the Servlet class is compiled to bytecode and loaded int o memory of the web server. At this point , the JSP is ready to deal with client requests. Execution starts when an HTTP request comes to the compiled Servlet. The service() method of the created Servlet gets called, which in turn calls the doGet() or doPost() method based on the request. The response is then generated and is usually an HTML document returned to the client's browser. If the JSP file is modified,
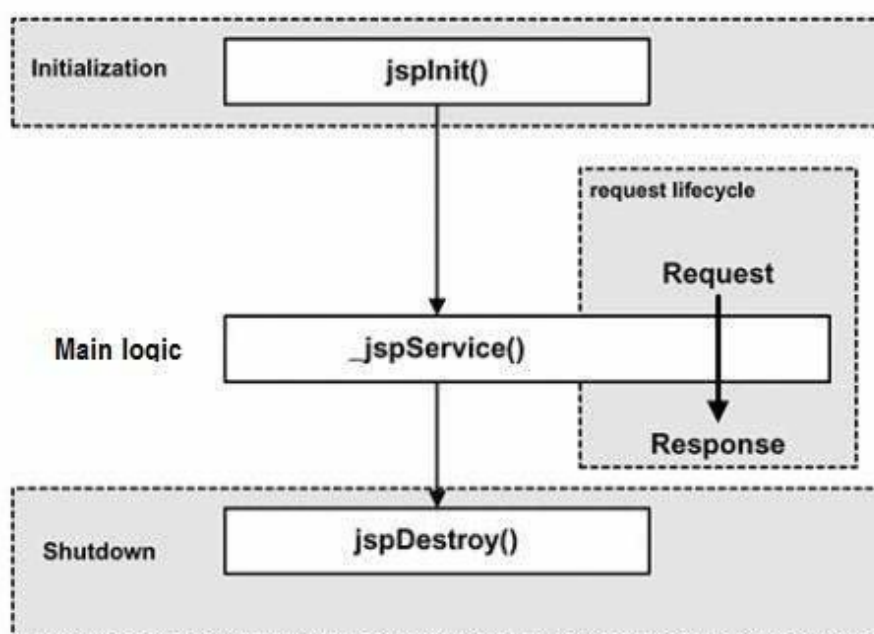
Figure 13.1: JSP Life Cycle

translation and compilation processes are restarted to account for changes.

13.5: Compilation, Execution, and Request Handling

After it is compiled int o a Servlet, execution of a JSP page is no different than that for a conventional Servlet. Initialization — the jspInit() method is called just once when the JSP page is visited for the first time. This is useful for configuring database connections or

initializing global application-wide variables. Next step is request processing (the jspService() method is invoked on each HTTP request) This approach collects the request parameters, accesses the business logic layer, and construct HTML content on the fly to serve a response. Destruction: Is the last step in the life cycle where before the JSP instance is removed from memory jspDestroy() method is called. This is handy for closing database connections, freeing resources, or doing cleanup tasks. JSP uses all the performance enhancements provided by Servlets (e.g. Caching, Session management) so it is a better technology in terms of building a scalable web app. The life cycle of a JSP is, therefore, essential to understanding how JSP-based applications can be optimized and how data is handled during request processing in actual production environments.

Check Your Progress

1. **Describe the lifecycle of a JSP page in detail.**

   ----

2. **Explain the advantages of JSP over servlet-based web development**.

13.6: Summary

JavaServer Pages (JSP) simplifies the creation of dynamic web applications by embedding Java code within HTML. Built on servlet technology, JSP automatically translates pages int o servlets for execution. Its scripting elements, directives, and expression language simplify server-side programming. By separating presentation from business logic, JSP improves maint ainability and scalability in enterprise web applications.

13.7: Exercises

Multiple Choice Questions

1. JSP stands for:

   a) Java Servlet Pages

   b) Java Server Pages

   c) Java Standard Pages

   d) Java Simple Pages

   Answer: b) Java Server Pages

2. Every JSP page is int ernally converted int o:

   a) JavaBean

   b) Servlet

   c) HTML file

   d) XML document

   Answer: b) Servlet

3. JSP technology is part of:

   a) Java SE

   b) Java EE

   c) Java ME

   d) JavaFX

   Answer: b) Java EE

4. JSP allows embedding of:

   a) SQL queries only

   b) HTML and Java code

   c) C++ syntax

   d) XML elements only

   Answer: b) HTML and Java code

5. Expression Language (EL) in JSP is used for:
   a) Compiling Java code
   b) Simplified data access
   c) Managing sessions
   d) Database transactions
   Answer: b) Simplified data access

Descriptive Questions

1. Explain the architecture and working of JSP.

2. Describe the lifecycle methods of a JSP page.

3. Compare servlet and JSP-based web applications.

4. Discuss the importance of Expression Language (EL) in JSP.

13.8 References and Suggested Readings

- Basham, B., Sierra, K., & Bates, B. (2014). *Head First Servlets and JSP*. O'Reilly Media.

- Hall, M. (2015). *Core Servlets and JavaServer Pages*. Pearson.

- Oracle (2025). *Jakarta JSP Specification*.

Block 4: JSP Technology

Unit 14: JSP Scripting Element and Directives

Structure

14.1: Introduction

JSP scripting elements and directives enable developers to embed Java code and control page behavior within JSP files. These elements provide the flexibility to insert logic directly int o the presentation layer while maint aining structured and reusable code. The three main scripting elements in JSP are:

1. **Declarations (<%! %>)** – used to declare variables and methods accessible to the entire JSP page.

2. **Scriptlets (<% %>)** – used to embed Java statements executed each time the page is requested.

3. **Expressions (<%= %>)** – used to output Java expressions directly int o the client's response.
   In addition to scripting elements, JSP supports **directives** that convey page-level instructions to the container. Common directives include:

4. **Page Directive** (<%@ page ... %>) – defines attributes like imports, error pages, or session handling.

5. **Include Directive** (<%@ include ... %>) – inserts static content from other files.

6. **Taglib Directive** (<%@ taglib ... %>) – imports custom tag libraries.
   These elements provide control over content generation, improve code reusability, and facilitate modular design. However, excessive use of scriptlets should be avoid ed in

favor of modern MVC-based approaches for cleaner separation of business and presentation logic.

14.2: Learning Outcomes

After completing this unit, learners will be able to:

1. Explain various JSP scripting elements.
2. Differentiate between declarations, scriptlets, and expressions.
3. Understand the purpose of JSP directives.
4. Apply include and taglib directives in JSP applications.
5. Design dynamic JSP pages using scripting and directive elements.

14.3: JSP Scripting Elements:

(i) JSP Declaration

- A declaration tag is a piece of Java code for declaring variables, methods and class es. If we declare a variable or method inside declaration tag it means that the declaration is made inside the servlet class but outside the service method.
- We can declare a static member, an instance variable (can declare a number or String) and methods inside the declaration tag.

Syntax of declaration tag:

<%! Dec var %>

Here Dec var is the method or a variable inside the declaration tag.

Example:

In this example, we are going to use the declaration tags

<%@ page language="java" contentType="text/html; char   set=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC   "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; char   set=ISO-8859-1">
<title>Guru Declaration Tag</title>
</head>

```
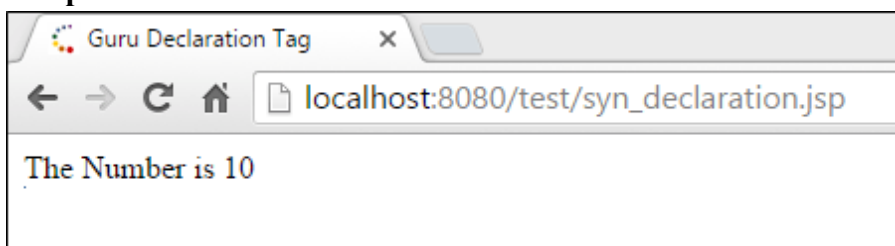<body>
<%! int  count =10; %>
<% out.print ln("The Number is " +count); %>
</body>
</html>
```

Explanation the code:

Code Line 10: Here we are using declaration tag for initializing a variable count to 10.

When you execute the above code you get the following output:

Output:

The variable which is declared in the declaration tag is print ed as output.

(ii) JSP Scriptlet

- Scriptlet tag allows to write Java code int o JSP file.
- JSP container moves statements in _jspservice() method while generating servlet from jsp.
- For each request of the client, service method of the JSP gets invoked hence the code inside the Scriptlet executes for every request.
- A Scriptlet contains java code that is executed every time JSP is invoked.

Syntax of Scriptlet tag:

`<% java code %>`

Here `<%%>` tags are scriplets tag and within it, we can place java code.

Example:

In this example, we are taking Scriptlet tags which enclose java code.

```
<%@ page language="java" contentType="text/html; char   set=ISO-8859-1"
  pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC            "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
```

```
<head>
<meta http-equiv="Content-Type" content="text/html; char   set=ISO-
8859-1">
<title>Guru Scriplet</title>
</head>
<body>
<% int  num1=10;
   int  num2=40;
   int  num3 = num1+num2;
   out.print ln("Scriplet Number is " +num3);
%>
</body>
</html>
```

Explanation of the code:

Code Line 10-14: In the Scriptlet tags where we are taking two variables num1 and num2 . Third variable num3 is taken which adds up as num1 and num2.The output is num3.

When you execute the code, you get the following output:

Output:

The output for the Scriptlet Number is 50 which is addition of num1 and num2.
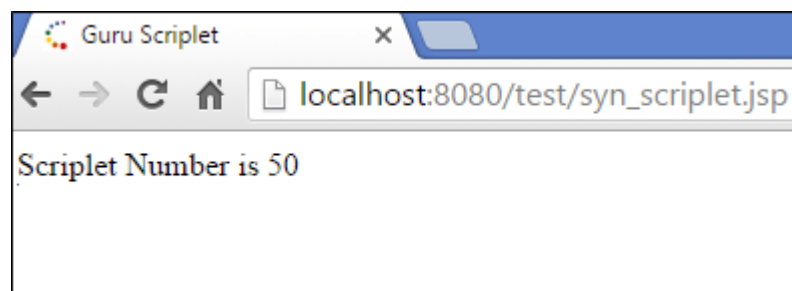
(iii) JSP Expression

- Expression tag evaluates the expression placed in it.
- It accesses the data stored in stored application.
- It allows create expressions like arithmetic and logical.
- It produces scriptless JSP page.

Syntax:

<%= expression %>

Here the expression is the arithmetic or logical expression.

Example:

In this example, we are using expression tag

```
<%@ page language="java" contentType="text/html; char   set=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC            "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; char   set=ISO-8859-1">
<title>Guru Expression</title>
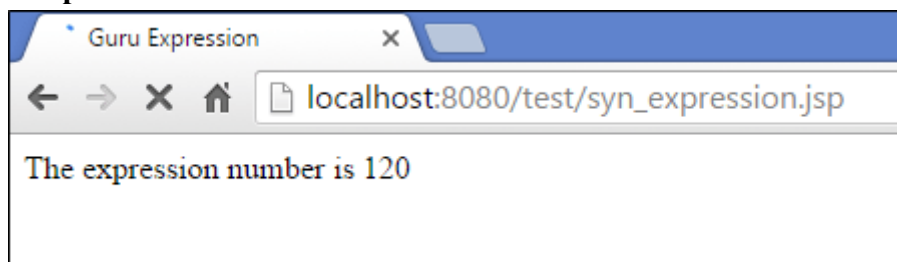</head>
<body>
<% out.print ln("The expression number is "); %>
<% int  num1=10; int  num2=10; int  num3 = 20; %>
<%= num1*num2+num3 %>
</body>
</html>
```

Explanation of the code:

Code Line 12: Here we are using expression tags where we are using an expression by multiplying two numbers i.e. num1 and num 2 and then adding the third number i.e. num3.

When you execute the above code, you get the following output:

Output:

The expression number is 120 where we are multiplying two numbers num1 and num2 and adding that number with the third number.

(iv) JSP Comments

Comments are the one when JSP container wants to ignore certain texts and statements.

When we want to hide certain content, then we can add that to the comments section.

Syntax:

<% -- JSP Comments %>

This tags are used to comment in JSP and ignored by the JSP container.

<!—comment –>

This is HTML comment which is ignored by browser

Example:

In this example, we are using JSP comments

<%@ page language="java" contentType="text/html; char set=ISO-8859-1"

   pageEncoding="ISO-8859-1"%>

<!DOCTYPE html PUBLIC   "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; char set=ISO-8859-1">

<title>Guru Comments</title>

</head>

<body>

<%-- Guru Comments section --%>

<% out.print ln("This is comments example"); %>

</body>

</html>

Explanation of the code:

Code Line 10: Here we are adding JSP comments to the code to explain what code has. It is been ignored by the JSP container

When you execute the above code you get the following output:

Output:

We get the output that is print ed in print ln method. Comments are ignored by container

14.4: Creating a simple JSP Page

A JSP page has an HTML body incorporated with Java code int o it

We are creating a simple JSP page which includes declarations, scriplets, expressions, comments tags in it.

Example:

```
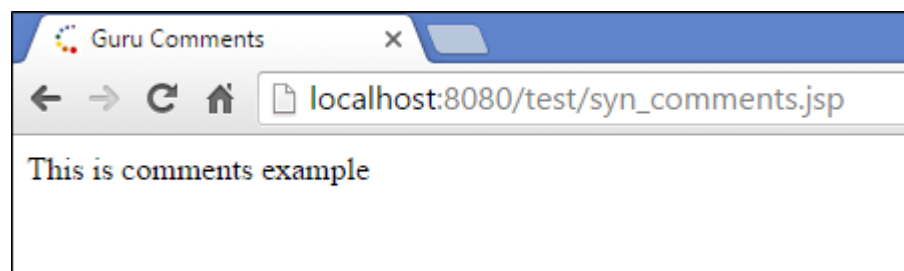<%@ page language="java" contentType="text/html; char  set=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC             "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; char  set=ISO-8859-1">
<title>Guru JSP Example</title>
</head>
<body>
<%-- This is a JSP example with scriplets, comments , expressions --%>
<% out.print ln("This is guru JSP Example"); %>
<% out.print ln("The number is "); %>
<%! int  num12 = 12; int  num32 = 12; %>
<%= num12*num32 %>
Today's date: <%= (new      java.util.Date()).toLocaleString  ()%>
</body>
</html>
```

Explanation of the code:

Code Line 1: Here we are using directives like language, contentType and pageEncoding. Language is Java and content type is text/html with standard char set ISO 8859. Page encoding is standard char set.

Code Line 11: Here we are using JSP comments to add comments to the JSP

Code Line 14: Here we are declaring variables num12 and num32 initializing with 12.

Code Line 15: Here we are using an expression where we are multiplying two numbers num12 and num32.

Code Line 16: Here we are fetching today's date using date object.

When you execute the above code, you get the following output

Output:
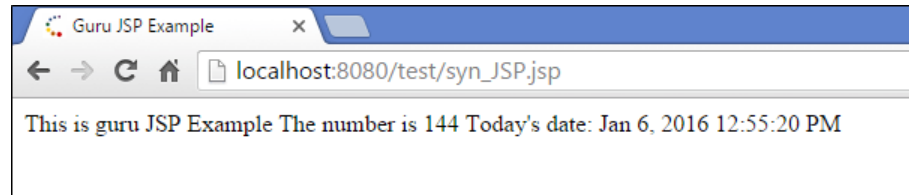
We are print ing overhere,

- This is guru JSP example.
- The number is num12*num32 (12*12).
- Today's date is the current date

14.5: How to run simple JSP Page

JSP can be run on web servers or application servers. Here we will be using a webserver, and we can deploy it on the server enclosing it in a war application. We can create JSP in an application (war).

This is an application which has following directory structure, and the application has to be build.
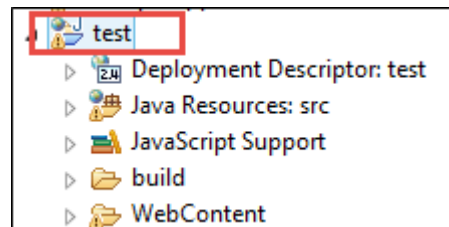
This application has to be built, and the following message will appear after the build is successful:
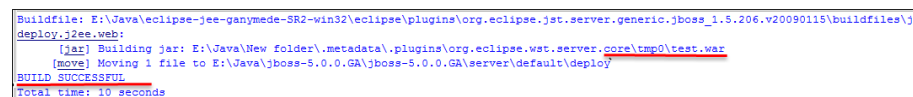
After the application is built then, the application has to be run on the server.

To run JSP on the webserver, right click on the project of the IDE (eclipse used in this case) and there are many options. Select the option of run on the server. It is shown in the screenshot below;

From the diagram, following point s are explained:

1. There are two options either to choose a server or manually add the server to this application. In this case, we have already added JBoss server to the application hence, we select the existing server.

2. Once we select the server the server option is shown in point 2 which server we want to select. There can be multiple servers configured on this application. We can select one server from all those options

Once that option is selected click on finish button and application will run on that server.
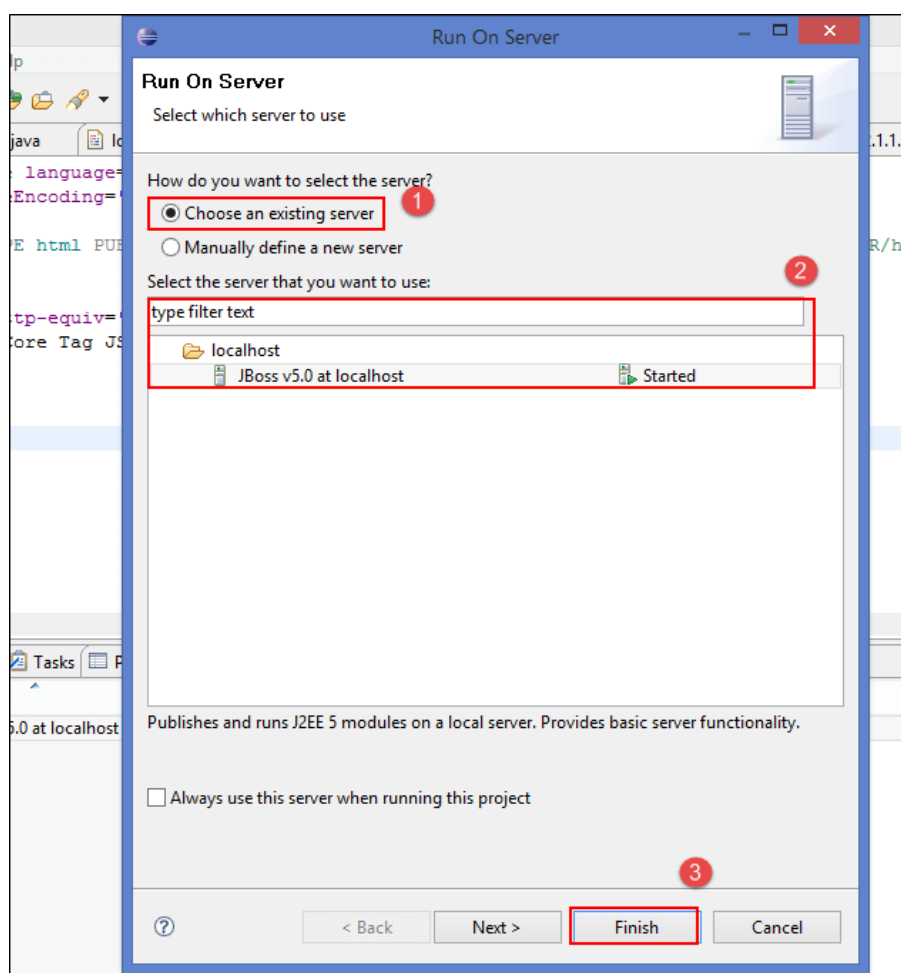
In the below screenshots, you can notice that our JSP program gets executed, and the test application is deployed in JBoss server marked in the red box.

Server	State	Status
JBoss v5.0 at localhost	Started	Synchronized
test		Synchronized

14.6: Directory Structure of JSP

In directory structure, there is a root folder which has folder WEB-INF, which has all configuration files and library files.

JSP files are outside WEB-INF folder

Example:

In this example there is test application which has folder structure has following:
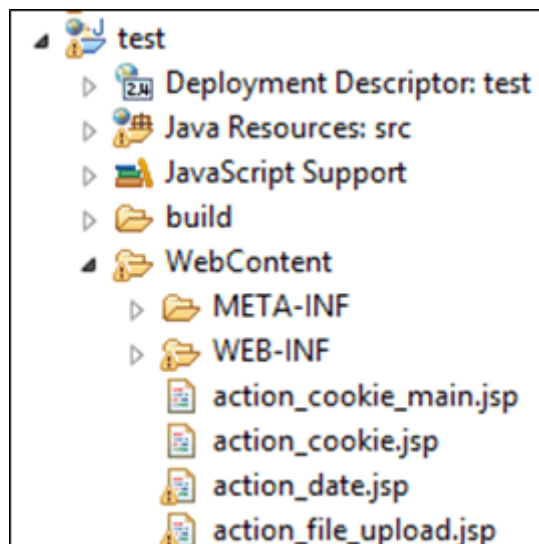
So JavaServer Pages (JSP) technology allows developers to create dynamic web content by embedding Java code within HTML pages. JSP scripting elements are the mechanisms through which this int egration occurs, enabling the execution of Java logic within the web page. These elements fall int o three primary categories: scriptlet tags, expression tags, and declaration tags. Scriptlet tags, denoted by <% ... %>, are used to embed Java code that will be executed when the JSP page is requested. This code can include any valid Java statement, such as variable declarations, control flow statements (if-else, loops), and method calls. Scriptlets are particularly useful for performing server-side processing, such as retrieving data from a database, manipulating data, and generating dynamic content. For instance, a scriptlet could be

used to iterate through a list of products and display them in an HTML table. Expression tags, represented by <%= ... %>, are used to insert the result of a Java expression directly int o the output stream. The expression within the tag is evaluated, and the result is converted to a String and inserted int o the HTML. This is useful for displaying dynamic data, such as the current date and time, user input, or the result of a calculation. For example, <%= new java.util.Date() %> would display the current date and time. Declaration tags, denoted by <%! ... %>, are used to declare variables and methods that are accessible throughout the JSP page. Declarations are typically placed at the beginning of the JSP page and are used to define reusable code components. For example, a declaration could be used to define a method that calculates the sum of two numbers, which can then be called from scriptlets or expression tags within the page. The order in which these scripting elements are processed is crucial. Scriptlets are executed first, followed by declarations, and then expressions. Scriptlets can modify the state of the page, such as by setting request or session attributes, which can then be accessed by subsequent scriptlets or expressions. Declarations define the structure of the JSP page, such as by defining variables and methods that can be used throughout the page. Expressions are evaluated and their results are inserted int o the output stream, generating the dynamic content that is displayed to the user. The use of scripting elements allows developers to create dynamic web pages that respond to user input and server-side events. However, excessive use of scriptlets can lead to code that is difficult to maint ain and debug. Best practices suggest minimizing the use of scriptlets and encapsulating business logic in Java class es, which can then be accessed from the JSP page using JavaBeans or custom tags. This approach promotes code reusability, maint ainability, and separation of concerns.Answer: JavaServer Pages (JSP) is a technology that helps software developers create dynamically-generated web pages based on HTML, XML, or other document types. This int egration occurs by way of JSP scripting elements, which allow Java logic to be executed inside the web page. These components are divided int o three types : scriptlet tags, expression tags and declaration tags. The flags denoted by are scriptlets which are hashed embedded Java Code that will be executed when the JSP page will be requested. Your code here can be any legal Java statement – variable declarations, control flow

statements (if-else, loops), or a call to a method. One of the special purposes where scriptlets can be very helpful is server-side processing in which it can be used to pull to data from the database, process data, and generate dynamic content. Example: Show products in an HTML table using Scriptlet But example: You can show the list of products in an HTML table using a Scriptlet. Expression tags () insert the result of a Java expression int o the output stream. It evaluates the expression found inside the tag, converts the result to a String , and inserts it in the HTML. This is also useful for showing dynamic data like current date/time, user input, or result of a calculation. For example, — would render the current date and time. These declaration tags start with is used to import any Java class es/page variables declared here are available throughout the JSP page. Declarations are used to define reusable code components and are normally found at the top of the JSP page. For purpose, a declaration will help you tell Jsp that it's a method that calculates the addition of 2 numbers and that method can also be called in Jsp via script lets or expression tags. The sequence for the processing of these scripting components is very important. The order of scriptlets, declarations, and expressions are executed one after the other. Scriptlets can change the state of the page, such as by setting request or session attributes, which can be read by subsequent scriptlets or expressions. Declarations specifies the structure of a JSP page by declaring a variables and methods, these can be used in the whole JSP page. This is done by evaluating the expressions and inserting their results int o the output stream, which is the dynamic content shown to the user as well. This is how to use scripting elements to develop dynamic web pages which respond and update showing user information and activities on the server. This doesn't always translate well when working with snippets of code, for example, in file processing or scrapers, where code is quickly written and deployed, sometimes in languages that require multiple steps to execute, such as Python. Recommended practice is to have a less use of scriptlet, keep the business logic in java class es and access these class es from the JSP page using JavaBeans or custom tags. It helps in reusing the code, maint ainability and separation.

Check Your Progress

1. **What are JSP scripting elements? Give examples**.

----

2. **Differentiate between page, include, and taglib directives**.

14.7: Summary

JSP scripting elements and directives allow embedding Java code and configuration within JSP pages. Declarations, scriptlets, and expressions support dynamic content creation, while directives manage page behavior and modular inclusion. Proper use of these elements enhances flexibility and functionality in JSP-based web development.

14.8: Exercise

Multiple Choice Question

1. JSP declarations are defined using:

    a) <% %>

    b) <%! %>

    c) <%= %>

    d) <%@ %>

    Answer: b) <%! %>

2. The <%@ page ... %> directive is used to:

    a) Declare variables

    b) Provide page-level instructions to the container

    c) Output expressions

    d) Import tag libraries

    Answer: b) Provide page-level instructions to the container

3. The <%@ include %> directive:

    a) Inserts static content at translation time

    b) Inserts dynamic content at runtime

    c) Includes external JAR files

d) Loads images dynamically

Answer: a) Inserts static content at translation time

4. The <%= %> tag is used for:

a) Declaring variables

b) Executing Java statements

c) Displaying results of expressions

d) Including pages

Answer: c) Displaying results of expressions

5. The directive used to include tag libraries is:

a) <%@ taglib %>

b) <%@ library %>

c) <%@ import %>

d) <%@ page %>

Answer: a) <%@ taglib %>

Descriptive Questions

1. Describe various JSP scripting elements with examples.
2. Explain the use of JSP directives and their types.
3. Write a JSP page using page and include directives.
4. Discuss best practices for using scriptlets in JSP applications

14.9: References and Suggested Readings

- Basham et al. (2014). *Head First Servlets and JSP*. O'Reilly.

- Hall, M. (2015). *Core Servlets and JSP*. Pearson.

- Oracle (2025). *Jakarta JSP Developer Guide*.

Block 4: JSP Technology

Unit 15: Implicit Object

Structure

15.1: Introduction

JSP provides several **implicit objects** that simplify int eraction with client requests, responses, sessions, and the application context without explicitly declaring them. These objects are automatically created by the JSP container and available to developers within the JSP page. The most commonly used implicit objects include:

- request: stores client request data.

- response: manages server responses.

- session: maint ains user-specific data across multiple requests.

- application: provides access to global application-level attributes.

- out: used to send output to the client.

- config, page, and exception provide configuration and error-handling support. In addition to these objects, JSP int roduced **Expression Language (EL)** to simplify access to data stored in JavaBeans, maps, or implicit objects using a concise syntax like ${object.property}. EL improves code readability by eliminating Java scriptlets in JSP pages. Using implicit objects and EL together enhances productivity by providing a cleaner, more declarative way of accessing server-side data and user sessions, thereby promoting a more maint ainable application design.

15.2: Learning Outcomes

After completing this unit, learners will be able to:

1. Identify and describe JSP implicit objects.
2. Access and manipulate data using implicit objects.
3. Explain the purpose and syntax of Expression Language (EL).
4. Use EL for accessing scoped variables.
5. Design JSP pages using implicit objects and EL effectively.

15.3: Implicit Objects

(i) What is JSP Implicit object?

- JSP implicit objects are created during the translation phase of JSP to the servlet.
- These objects can be directly used in scriplets that goes in the service method.
- They are created by the container automatically, and they can be accessed using objects.

(ii) How many Implicit Objects are available in JSP?

There are 9 types of implicit objects available in the container:

1. Out
2. Request
3. Response
4. Config
5. Application
6. Session
7. PageContext
8. Page
9. Exception

1) Out: Out is one of the implicit objects to write the data to the buffer and send output to the client in response,Out object allows us to access the servlet's output stream, Out is object of javax.servlet.jsp.jspWriter class , While working with servlet, we need print writer object

Example:

```
<%@ page language="java" contentType="text/html; char   set=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
```

```
<!DOCTYPE html PUBLIC              "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; char   set=ISO-
8859-1">
<title>Implicit Guru JSP1</title>
</head>
<body>
<% int  num1=10;int  num2=20;
out.print ln("num1 is " +num1);
out.print ln("num2 is "+num2);
%>
</body>
</html>
```

Explanation of the code:

Code Line 11-12– out is used to print int o output stream

When we execute the above code, we get the following output:

Output:

- In the output, we get the values of num1 and num2

2) Request

- The request object is an instance of java.servlet.http.HttpServletRequest and it is one of the argument of service method
- It will be created by container for every request.
- It will be used to request the information like parameter, header information , server name, etc.
- It uses getParameter() to access the request parameter.

Example:

Implicit_jsp2.jsp(form from which request is sent to guru.jsp)

<%@ page language="java" contentType="text/html; char set=ISO-8859-1"

  pageEncoding="ISO-8859-1"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; char set=ISO-8859-1">

<title>Implicit Guru form JSP2</title>

</head>

<body>

<form action="guru.jsp">

<input type="text" name="username">

<input type="submit" value="submit">

</form>

</body>

</html>

Guru.jsp (where the action is taken)

```
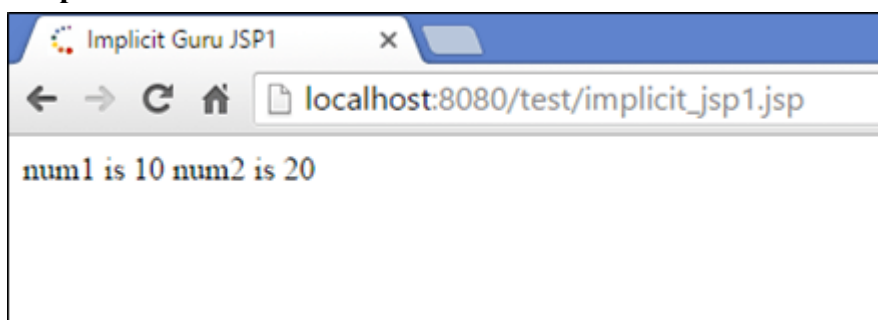1  <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2      pageEncoding="ISO-8859-1"%>
3  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
4  <html>
5  <head>
6  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
7  <title>form action</title>
8  </head>
9  <body>
10 <% String username = request.getParameter("username");
11 out.println("Welcome " +username);
12 %>
13 </body>
14 </html>
```

Explanation of code:

Code Line 10-13 : In implicit_jsp2.jsp(form) request is sent, hence the variable username is processed and sent to guru.jsp which is action of JSP.

Guru.jsp

Code Line10-11: It is action jsp where the request is processed, and username is taken from form jsp.

When you execute the above code, you get the following output
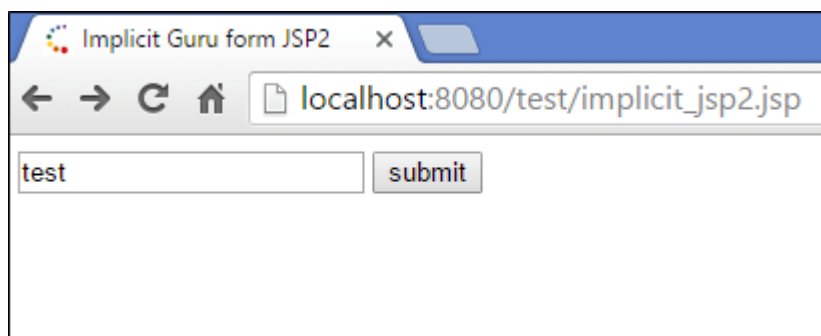
Output:

When you write test and click on the submit button, then you get the following output "Welcome Test."
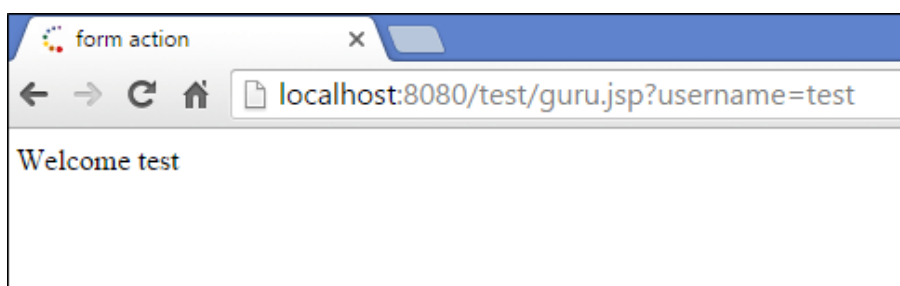
3) Response

- "Response" is an instance of class which implements HttpServletResponse int erface
- Container generates this object and passes to _jspservice() method as parameter
- "Response object" will be created by the container for each request.
- It represents the response that can be given to the client
- The response implicit object is used to content type, add cookie and redirect to response page

Example:

```
<%@ page language="java" contentType="text/html; char  set=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC            "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
```

<meta http-equiv="Content-Type" content="text/html; char set=ISO-8859-1">

<title>Implicit Guru JSP4</title>

</head>

<body>

<%response.setContentType("text/html"); %>

</body>

</html>

Explanation of the code:

Code Line 11: In the response object we can set the content type

Here we are setting only the content type in the response object. Hence, there is no output for this.

4) Config

- "Config" is of the type java.servlet.servletConfig
- It is created by the container for each jsp page
- It is used to get the initialization parameter in web.xml

Example:

Web.xml (specifies the name and mapping of the servlet)

```xml
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
3      <display-name>
4      test</display-name>
5      <servlet>
6          <description>
7          </description>
8          <display-name>
9          GuruServlet</display-name>
10         <servlet-name>GuruServlet</servlet-name>
11         <servlet-class>
12         demotest.GuruServlet</servlet-class>
13     </servlet>
14     <servlet-mapping>
15         <servlet-name>GuruServlet</servlet-name>
16         <url-pattern>/GuruServlet</url-pattern>
17     </servlet-mapping>
18     <welcome-file-list>
19         <welcome-file>index.html</welcome-file>
20         <welcome-file>index.htm</welcome-file>
21         <welcome-file>index.jsp</welcome-file>
22         <welcome-file>default.html</welcome-file>
23         <welcome-file>default.htm</welcome-file>
24         <welcome-file>default.jsp</welcome-file>
25     </welcome-file-list>
26 </web-app>
27
```

Implicit_jsp5.jsp (getting the value of servlet name)

<%@ page language="java" contentType="text/html; char set=ISO-8859-1"

  pageEncoding="ISO-8859-1"%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; char set=ISO-8859-1">

<title>Implicit Guru JSP5</title>

</head>

<body>

<% String   servletName = config.getServletName();

out.print ln("Servlet Name is " +servletName);%>

</body>

</html>

Explanation of the code:

In web.xml

Code Line 14-17: In web.xml we have mapping of servlets to the class

es.

Implicit_jsp5.jsp

Code Line 10-11: To get the name of the servlet in JSP, we can use config.getServletName, which will help us to get the name of the servlet.

When you execute the above code you get the following output:

- Servlet name is "GuruServlet" as the name is present in web.xml

5) Application

- Application object (code line 10) is an instance of javax.servlet.ServletContext and it is used to get the context information and attributes in JSP.
- Application object is created by container one per application, when the application gets deployed.
- Servletcontext object contains a set of methods which are used to int eract with the servlet container.We can find information about the servlet container

Example:

<%@ page language="java" contentType="text/html; char   set=ISO-8859-1"

   pageEncoding="ISO-8859-1"%>

<!DOCTYPE html PUBLIC   "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; char   set=ISO-8859-1">

```
<title>Guru Implicit JSP6</title>
</head>
<body>
<% application.getContextPath(); %>
</body>
</html>
```

Explanation of the code:

• In the above code, application attribute helps to get the context path of the JSP page.

6) Session

- The session is holding "httpsession" object(code line 10).
- Session object is used to get, set and remove attributes to session scope and also used to get session information

Example:

Implicit_jsp7(attribute is set)

```
<%@ page language="java" contentType="text/html; char set=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC        "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; char set=ISO-8859-1">
<title>Implicit JSP</title>
</head>
<body>
<% session.setAttribute("user","GuruJSP"); %>
<a href="implicit_jsp8.jsp">Click here to get user name</a>
</body>
</html>
```

Implicit_jsp8.jsp (getAttribute)

```
<%@ page language="java" contentType="text/html; char set=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
```

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; char set=ISO-8859-1">
<title>implicit Guru JSP8</title>
</head>
<body>
<% String   name = (String)session.getAttribute("user");
out.print ln("User Name is " +name);
%>
</body>
</html>

Explanation of the code:

Implicit_jsp7.jsp

Code Line 11: we are setting the attribute user in the session variable, and that value can be fetched from the session in whichever jsp is called from that (_jsp8.jsp).

Code Line 12: We are calling another jsp on href in which we will get the value for attribute user which is set.

Implicit_jsp8.jsp

Code Line 11: We are getting the value of user attribute from session object and displaying that value

When you execute the above code, you get the following output:
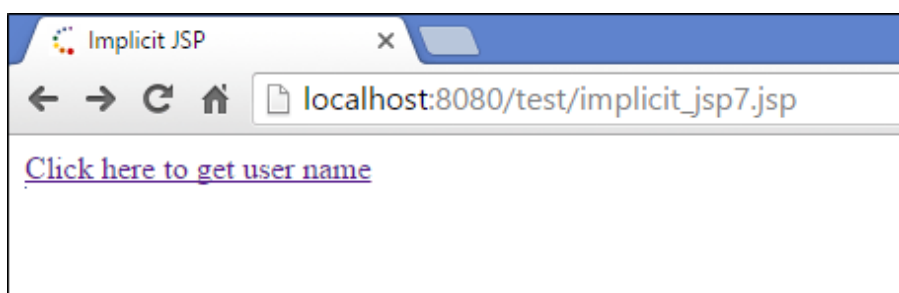
When you click on the link for the username. You will get the following output.

Output:
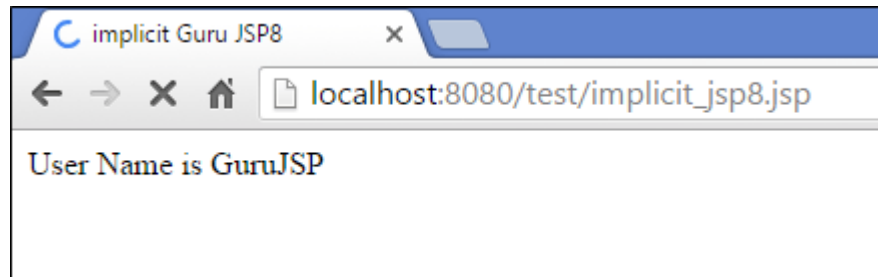
- When we click on link given in implicit_jsp7.jsp then we are redirected to second jsp page, i.e (_jsp8.jsp) page and we get the value from session object of the user attribute (_jsp7.jsp).

7) PageContext

- This object is of the type of pagecontext.
- It is used to get, set and remove the attributes from a particular scope

  Scopes are of 4 types:
- Page
- Request
- Session
- Application

Example:

```
<%@ page language="java" contentType="text/html; char   set=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC            "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; char   set=ISO-8859-1">
<title>Implicit Guru JSP9</title>
</head>
<body>
<%
pageContext.setAttribute("student","gurustudent",pageContext.PAGE_SCOPE);
String   name = (String  )pageContext.getAttribute("student");
out.print ln("student name is " +name);
```

%>

</body>

</html>

Explanation of the code:

Code Line 11: we are setting the attribute using pageContext object, and it has three parameters:

- Key
- Value
- Scope

In the above code, the key is student and value is "gurustudent" while the scope is the page scope. Here the scope is "page" and it can get using page scope only.

Code Line 12: We are getting the value of the attribute using pageContext

When you execute the above code, you get the following output:
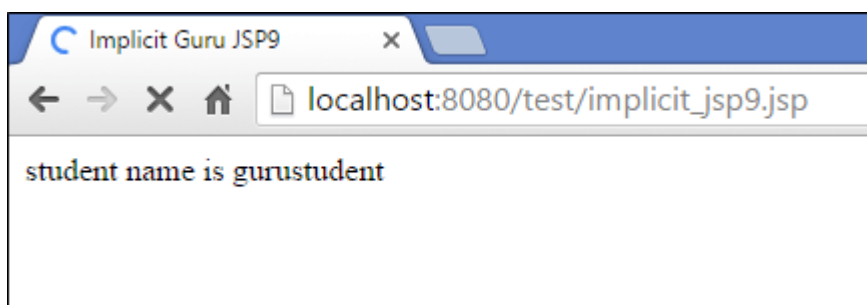
8) Page

- Page implicit variable holds the currently executed servlet object for the corresponding jsp.
- Acts as this object for current jsp page.

Example:

In this example, we are using page object to get the page name using toString   method

```
<%@ page language="java" contentType="text/html; char   set=ISO-
8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC              "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
```

MATS Centre for Distance and Online Education, MATS University

```
<meta http-equiv="Content-Type" content="text/html; char   set=ISO-
8859-1">
<title>Implicit Guru JSP10</title>
</head>
<body>
<% String   pageName = page.toString  ();
out.print ln("Page Name is " +pageName);%>
</body>
</html>
```

Explanation of the code:

Code Line 10-11: In this example, we are trying to use the method toString () of the page object and trying to get the String   name of theJSP Page.

When you execute the code you get the following output:

9) Exception
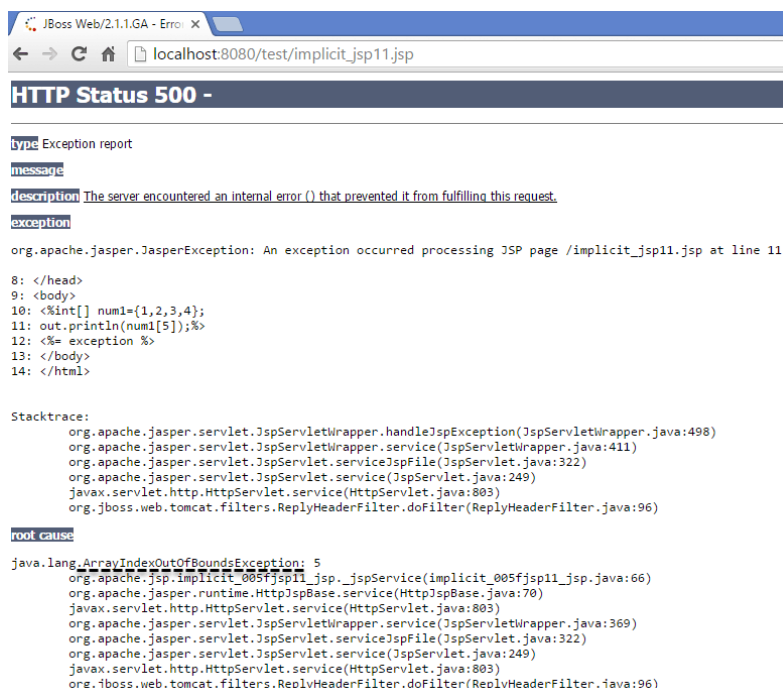
- Exception is the implicit object of the throwable class .
- It is used for exception handling in JSP.
- The exception object can be only used in error pages.

Example:

```
<%@ page language="java" contentType="text/html; char   set=ISO-
8859-1"
   pageEncoding="ISO-8859-1" isErrorPage="true"%>
<!DOCTYPE  html  PUBLIC            "-//W3C//DTD  HTML  4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
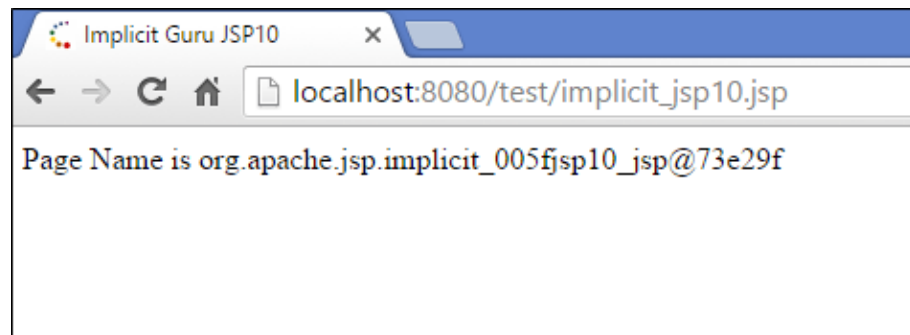<meta http-equiv="Content-Type" content="text/html; char   set=ISO-
8859-1">
<title>Implicit Guru JSP 11</title>
</head>
```

\<body>

\<%int [] num1={1,2,3,4};

out.print ln(num1[5]);%>

\<%= exception %>

\</body>

\</html>

Explanation of the code:Code Line 10-12 – It has an array of

numbers, i.e., num1 with four elements. In the output, we are trying to print the fifth element of the array from num1, which is not declared in the array list. So it is used to get exception object of the jsp.We are getting ArrayIndexOfBoundsException in the array where we are getting a num1 array of the fifth element.So JSP provides a set of predefined objects, known as implicit objects, which are automatically available to developers within the JSP page. These objects provide access to server-side resources and contextual information, simplifying the development of dynamic web applications. The implicit objects include request, response, config, application, session, pageContext, page, and exception. The request object, an instance of javax.servlet.http.HttpServletRequest, provides access to information about the client's request, such as request parameters, headers, and cookies. Developers can use the request object to retrieve form data, access session attributes, and handle file uploads. The response object, an instance of javax.servlet.http.HttpServletResponse, allows

developers to send data back to the client, such as HTML content, images, and other resources. Developers can use the response object to set response headers, cookies, and redirect the client to another page. The config object, an instance of javax.servlet.ServletConfig, provides access to servlet configuration information, such as initialization parameters and servlet context. Developers can use the config object to retrieve configuration settings for the JSP page. The application object, an instance of javax.servlet.ServletContext, provides access to application-wide resources and attributes. Developers can use the application object to share data between different JSP pages and servlets within the same web application. The session object, an instance of javax.servlet.http.HttpSession, provides access to session-specific data and attributes. Developers can use the session object to store user-specific information, such as login credentials and shopping cart contents. The pageContext object, an instance of javax.servlet.jsp.PageContext, provides access to the JSP page's context, including access to other implicit objects and page-scoped attributes. Developers can use the pageContext object to forward requests to other pages, include other resources, and manage page-scoped attributes. The page object, an instance of java.lang.Object, represents the JSP page itself. In most cases, it is equivalent to the this keyword. The exception object, an instance of java.lang.Throwable, is available only in error pages and provides access to the exception that caused the error. Developers can use the exception object to display error messages and log error details. The implicit objects are automatically created and initialized by the JSP container when the JSP page is requested. They are accessible within scriptlets, expression tags, and declaration tags. The request and session objects are particularly useful for managing user sessions and handling form data. The application object is useful for sharing data between different parts of the web application. The pageContext object provides a convenient way to access other implicit objects and manage page-scoped attributes. The exception object simplifies error handling in JSP pages. Understanding and effectively using these implicit objects is essential for developing robust and efficient JSP applications.

15.4: Directive Elements

Directive Elements JSP directive elements used to control the overall behavior of the JSP page and also to provide the configuration information to the JSP page to the container. These elements do not produce any output that has to be sent to the client, instead they control the general structure and behavior of the JSP page. Directive elements are found at the top of the JSP page and start with . Directive elements can be of three types: page, include, and taglib. Indeed, the page directive defines page-specific properties like content type, import statements, and error page configuration. The page directive contain attributes like contentType, import, errorPage, isErrorPage, session, buffer, autoFlush, info, isThreadSafe, language, extends. The contentType is a String representation of the MIME type of the response, e.g., text/html or application/json. The import attribute allows for the importing of Java class es and packages so that they are available for use in the JSP page. The URL of the error page to be displayed in case of Exception is defined using the errorPage attribute. isErrorPage attribute determines whether the page is an error page Example of using session in in JSP page The session attribute: Determines whether the JSP page participates in a session. ParseBuffer(buffer,20); This instruction parses a response buffer of 20 bytes. autoFlush attribute specifies the buffer autoFlush or not The info attribute provides a description of the JSP page. The isThreadSafe attribute indicates if the JSP page is thread safe. The language parameter specifies the scripting language in the JSP page. The extends attribute in id extends the superclass of the generated servlet. The line with the include directive looks like this: Other than that, the included file can be a static HTML file, another JSP, servlet or any other resource that is available to the JSP container. There are two forms of the include directive: a static one and a dynamic one. Static include: () – Includes the file at translation time, that is, the included file is processed only once, during JSP page compilation. Dynamic include or generates the file at request time which means the included file will be processed each time the JSP page is requested. This article explains the usage of JSP Taglibs along with an example JSP page. The taglib directive has two attributes: prefix and uri. The uri attribute indicates the URI of the tag library descriptor (TLD) file that defines the custom tags. The prefix attribute defines the prefix to be used by the

custom tags in the JSP page. At the same time, directive elements can guide the JSP pages in behavior and structure. They allows us to process page-level settings (like external resources) and add a custom tag. So keyword such are Directive element must be used appropriately in JSP application to more effectively.

15.5: Advanced JSP Scripting and Implicit Object Utilization

While its basic usage—combining JSP scripting elements and implicit objects—serves most purposes, advanced techniques can help optimise the functionality and efficiency of JSP applications significantly. For example, scriptlets can be utilized to execute complicated business logic like data validation, which involves checking data int egrity and accuracy against specific criteria, form processing can process user input from HTML forms to operate on, and database int eractions can fetch data from a database. Using scriptlets for presentation logic should be minimized, as it may cause code that is hard to maint ain and debug.

Directive Elements:

In this Article JSP (JavaServer Pages) directive elements define essential construction information for the JSP container regarding the information, dependencies and handling requirements of a webpage. These are not included in the output instead they are configuration directives that guide how the JSP page will be translated and executed. 1 There are three primary directive elements: the page directive, the include directive, and the taglib directive. You can give page specific information using this directive like content type of the page, how to handle the error for the page and about session management. It appears at the start of a JSP page and can consist of several attributes. The contentType property carries the MIME type and char acter encoding of the response, so that the client browser int erprets the response. Example: → This sets the content type as HTML with UTF-8 encoding. In this example, the errorPage attribute defines the URL of an error page to be displayed in case of an exception as part of that page processing. This gives a chance to handle errors gracefully & prevents users from having the raw stack traces. The isErrorPage property informs whether in the current context an error page is present, making possible the implementation of conditional error handling

logic. Session AttributeExecutes on endInitialize | endLoadSyntaxPage Attributes When set to true, the session attribute allows or prevents SQL session management for the page. If set to true, the session implicit object will be available as well, enabling developers to access the session data. Developers can import Java class es and packages, which become available for use in the JSP page by using the import attribute. For multiple import attributes multiple class es or packages can be imported. The language attribute also specifies the scripting language of the JSP page which is usually Java. Other properties, like buffer, autoFlush, and info, offer more fine-grained control over the process of paging. The include directive allows you to include a file in the JSP page at translation time. It enables code reuse and modular development. The file to include could be a JSP page, HTML file, or any text file. The file attribute defines the path to the file to be included. For example, // contains master header jsp file. Since the included file is processed just like part of the current page, any changes to the included file would cause the JSP page to be recompiled. 2 The taglib directive is used to declare a tag library so that its custom tags can be used in this JSP page. They offer a way to encapsulate commonly-used functionality and make JSP development simpler. 3 The uri attribute declares the URI of the tag library and the prefix attribute declares a prefix to identify in the library the tags. For example, declares the JSTL core tag library with prefix c. After declaring a tag library, its custom tags can be used in the JSP page using the specified prefix within the JSP page. 4 It is three-line configuration. Because they perform the directives, which control the behaviour of JSP, they allow page authors to have more control over their JSP

15.6: Action Elements

JSP action elements are runtime instructions that dynamically generate content or control the flow of execution within a JSP page. Unlike directive elements, which are processed at translation time, action elements are executed at runtime, allowing for dynamic behavior. The two primary action elements are jsp:forward and jsp:include, each serving distinct purposes in JSP development. The jsp:forward action element is used to transfer control from the current JSP page to another resource, such as another JSP page, servlet, or HTML file. It effectively

redirects the request to the specified resource, and the current page ceases processing. The page attribute specifies the relative or absolute URL of the resource to which control should be transferred. For instance, <jsp:forward page="welcome.jsp" /> forwards the request to the welcome.jsp page. The jsp:forward action can also include parameters using the jsp:param sub-element, allowing developers to pass data to the target resource. For instance, <jsp:forward page="profile.jsp"><jsp:param name="userId" value="123" /></jsp:forward> forwards the request to the profile.jsp page with the userId parameter set to 123. The jsp:forward action is often used for implementing navigation logic, error handling, and conditional page flow. It is crucial to note that once the jsp:forward action is executed, any output buffered by the current page is discarded, and the response is generated by the target resource. The jsp:include action element is used to include the output of another resource int o the current JSP page at runtime. This allows for dynamic content inclusion and modular development. The page attribute specifies the relative or absolute URL of the resource to be included. For instance, <jsp:include page="footer.jsp" /> includes the output of the footer.jsp page. The included resource is executed, and its output is inserted int o the response stream of the current page. The jsp:include action can also include parameters using the jsp:param sub-element, allowing developers to pass data to the included resource. For instance, <jsp:include page="new   s.jsp"><jsp:param name="category" value="sports" /></jsp:include> includes the output of the new   s.jsp page with the category parameter set to sports. The jsp:include action is often used for including common page elements, such as headers, footers, and navigation bars, dynamically. It allows for creating reusable components and maint aining consistency across multiple pages. Unlike the include directive, which includes files at translation time, the jsp:include action includes resources at runtime, allowing for dynamic content generation. Action elements provide a powerful mechanism for controlling the flow of execution and generating dynamic content within JSP pages, enabling developers to create int eractive and dynamic web applications.JSP action elements are instructions that are executed during runtime and are used to dynamically generate content or control the flow of execution in a JSP page. Whereas directive elements are processed during the translation

phase, action elements are executed in the runtime phase, providing dynamic run-time behavior. The only two dominant action elements are jsp:forward and jsp:include and they serve different purposes. jsp:forward action element Transfers control from one JSP page to another JSP page, servlet, or HTML file. This is useful as it makes use of the request and is placed on the JSP page itself. It does its job of re-routing the request to the targeted resource and the current page stops its processing. The page attribute indicates the relative or absolute URL of the resource to which control will be transferred. Such as forwards the request to welcome. jsp page. jsp:forward action may also pass parameters to the target resource with the jsp:param sub-element, thus, developers can also pass some data to the target resource. Example, forwards the request to the profile. The Web/cgi-bin/launchpage.jsp page with the userId parameter set to 123. Also, jsp:forward action is commonly used for navigation logic, error handling and conditional page flow. Please note that, upon executing the jsp:forward action, anything that is output buffered by the current page will be: discarded and the response will be generated by the target resource. JSP JSP:include The jsp:include action element is used to include the output of another resource (servlet, JSP file, etc) in the current JSP page at runtime. Dynamic content inclusion and modular development. The page property point s to the relative or absolute URL of the page to include For example, outputs the footer. jsp page. The included resource is invoked and the result is inserted directly int o the response stream such that it becomes part of the output of the current page. The developer of the included resource must access the included resource through the request object just as with the request, but the developer of the included resource can also pass parameters if they exist within it as sub-elements to the parent include. For example, includes the output of the new s jsp page — the category parameter set to sports. jsp:include action is frequently utilized to dynamically include shared components like headers, footers, or navigation bars. It enables the development of reusable components and the seamless preservation of uniformity across different pages. The jsp:include action differs from the include directive in that the include directive includes files at translation time, whereas the jsp:include action includes resources at runtime, enabling dynamic content generation. By acting as a combination of XML and Java, action elements are a great way to

control your flow of execution and generate dynamic content within JSP pages.

15.7: Page Directive: Configuring Page-Specific Attributes

The page directive in JSP development is one of the primary methods through which a developer can define several things on a page that affect the way the JSP container manages this page. Syntax : It is usually found at the top of a JSP page and it has one or more attributes each of which has its own purpose. Here the contentType attribute is used to state the mime type of JSP page response and the char acter encoding. This attribute makes sure that the client browser understand the content. For example: defines that the content type is HTML, encoded in UTF-8, so the page will render the HTML content encoded in UTF-8 Some other widely used values are text/plain, application/json, and application/xml, according to the content being produced. On imports tag JSP developers can make use of Java class es and packages in JSP page. It makes development JSP so simpler because you will not have to use fully qualified class names. Import multiple class es or packages using import for multiple import attributes e.g. imports all class es in the java. util package. If an exception occurs and the errorPage attribute of the page is specified, the page URL specified in errorPage will be invoked. That means we can implement graceful error handling and avoid raw stack traces from being displayed to users. For example, If there is any Exception then a jsp page should be shown. Check whether the current page is an error page with the isErrorPage attribute This post is related to the exception implicit object that is available when the isException=true. For example, specifies that the page in question is an error page. The session attribute is used to enable or disable session management for the page. When this is true, the session implicit object is made available and developers can store and read session data. Example: The creates session management for the page. This uses a buffer attribute where you can set the buffer size for the output stream before writing it to the client autoFlush Specifies whether the buffer will be automatically flushed if the buffer is full. The info attribute is a String storing a description of the page, which can be obtained through calling the HttpServlet class getServletInfo() method. The language attribute

defines the scripting language that's used in a JSP page, it is Java in usual. The other attributes that can be specified (extends, pageEncoding and isThreadSafe) give more control over how the page is processed. It is important to note that the page directive is critically important in setting up page-specific information so that the JSP container can process the page as per the information given.

15.8: Include Directive

One is the include directive, which is a powerful tool in JSP development, allowing developers to insert the content of another file int o the current JSP page at translation time. This allows for modular development and code reuse, as common elements that appear on multiple pages can be factored out int o separate files and then included in multiple JSP pages. This last parameter is the file to be included. The path can be a relative/absolute path depending upon where the included file is.

Check Your Progress

1. **Explain the role of JSP implicit objects with examples.**

   ----

2. **Describe how Expression Language (EL) simplifies JSP development.**

15.9 Summary

This module int roduces learners to JavaServer Pages (JSP), a powerful server-side technology used to create dynamic web content. It begins by explaining the concept, need, and advantages of JSP over traditional servlet-based programming. JSP allows embedding Java code directly int o HTML, making web development more efficient and content

easier to manage.Learners explore the JSP life cycle, which includes stages like translation, compilation, initialization, execution, and destruction.

Understanding this life cycle helps in managing how JSP pages are processed and served by the web server.The module also covers various scripting elements—such as declarations, expressions, and scriptlets—which are used to insert Java code int o JSP pages. Additionally, it int roduces implicit objects like request, response, session, application, and out, which simplify coding by giving direct access to common functionalities. Further, the module dives int o directive elements (e.g., page, include, taglib) and action elements (e.g., <jsp:include>, <jsp:forward>, <jsp:useBean>) that control page behavior and enhance dynamic content management.By the end of this module, learners gain a comprehensive understanding of JSP development, enabling them to build robust, dynamic, and maint ainable web applications.

15.10: Exercises

Multiple Choice Questions

1. Which implicit object represents the client's request?

   a) session

   b) request

   c) application

   d) config

   Answer: b) request

2. The implicit object used to write output to the client is:

   a) out

   b) response

   c) writer

   d) output

   Answer: a) out

3. The session object is used to:

   a) Store per-user data between requests

   b) Store global application data

   c) Access web.xml parameters

   d) Manage cookies only

   Answer: a) Store per-user data between requests

4. EL expressions in JSP are written as:

   a) <% expression %>

   b) ${expression}

   c) <%= expression %>

   d) <el:expression>

   Answer: b) ${expression}

5. The application implicit object represents:

   a) The servlet configuration

   b) The global servlet context

   c) The request parameters

   d) The session attributes

   Answer: b) The global servlet context

Descriptive Questions

1. Explain various JSP implicit objects and their purposes.
2. Discuss how Expression Language improves JSP readability.
3. Write a JSP page demonstrating the use of EL with session attributes.
4. Compare scriptlet-based and EL-based data access in JSP.

15.11 References and Suggested Readings

- Basham et al. (2014). *Head First Servlets and JSP*. O'Reilly.

- Hall, M. (2015). *Core Servlets and JSP*. Pearson.

- Oracle (2025). *Jakarta JSP Specification*.

Block 5: Spring and Spring Boot Framework

Unit 16: Introduction to Spring Initializing and Writing Spring Application

Structure

16.1: Introduction

Spring is a powerful, lightweight, and modular Java framework that simplifies the development of enterprise applications by promoting loose coupling and high testability through Dependency Injection (DI) and Aspect-Oriented Programming (AOP). It provides a comprehensive infrastructure for building web, microservice, and data-driven applications efficiently. The framework allows developers to focus on business logic while managing system-level concerns like configuration, transactions, and security. Spring's modular architecture enables developers to use only the necessary components, making applications lightweight and maint ainable. Spring Boot further enhances productivity by providing auto-configuration, embedded servers, and production-ready defaults. Together, Spring and Spring Boot streamline the process of initializing, configuring, and deploying modern Java applications.

16.2: Learning Outcomes

After completing this unit, learners will be able to:

1. Explain the core concepts and principles of the Spring Framework, including Dependency Injection (DI) and Aspect-Oriented Programming (AOP).

2. Initialize and configure a Spring application using XML-based, annotation-based, and Java-based approaches.

3. Develop Spring applications by creating and managing beans with annotations such as @Component, @Service, @Repository, and @Controller.

4. Implement Dependency Injection to promote loose coupling and improve modularity and reusability of components.

5. Use Spring Boot for building stand-alone, production-ready applications with minimal configuration and setup.

6. Apply best practices for writing, testing, and deploying Spring-based web and enterprise applications efficiently

16.3: Spring Initializing

Spring, a full-fledged and well-accepted framework that has absolutely changed the way Java applications are developed by providing an infrastructure to develop enterprise applications. Spring is a container framework that is designed to develop very loosely coupled easily testable and maint ainable applications based on DI(AOP) principles under the hood. Spring is designed in a modular way, meaning developers can pick and choose only the aspects that they will need, making it a light development environment. The framework is capable of serving different types of applications like web applications, microservices, and batch processing systems. Spring was conceived out of a desire to overcome the challenges and confines of Java EE, providing a more agile and pragmatic approach to application development. Over the years, the framework has evolved to support new technologies and methodologies, making it a popular choice among developers. Spring is a collection of many different modules that focus on different aspects of application development. The heart of this framework is its core container, responsible for managing the full lifecycle of application components (beans). The Spring's DI mechanism helps developers configure the dependencies for beans and process these beans by injecting the dependencies for them in runtime. By doing so, we encourage loose coupling, such that int erdependencies between code are reduced and code is more reusable. Aspect-Oriented Programming, or AOP, is another key pillar of Spring, offering a way to modularize cross-cutting concerns like logging, security, and transaction management. Aspects can also handle cross-cutting concerns, allowing developers to encapsulate these concerns int o facets that can be applied uniformly to the application without

polluting the business logic itself. The Spring framework enables seamless int eraction with different data access technologies like JDBC,
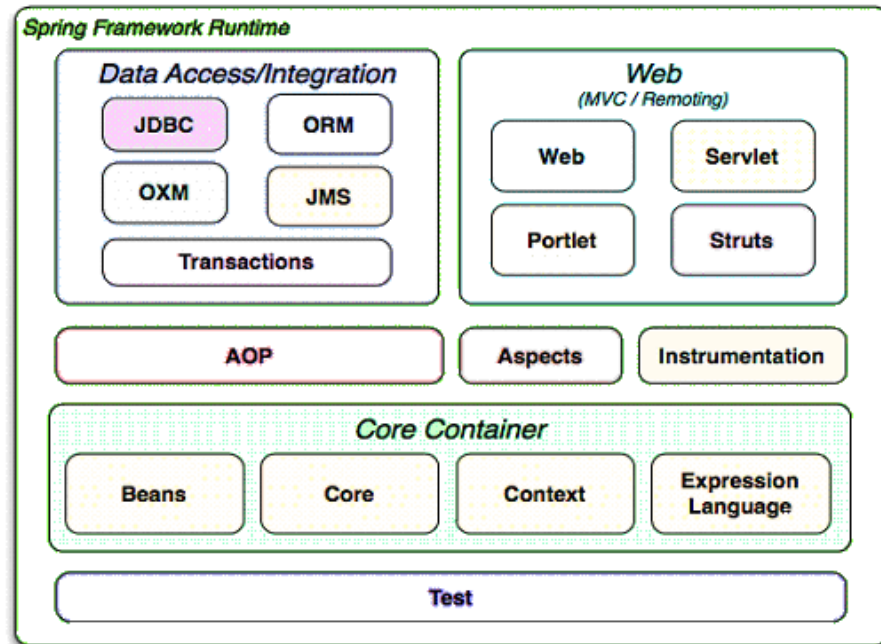
Figure 16.1: Spring Framework

project of Spring that has taken the core components of Spring and provided sensible defaults for creating stand-alone, production-ready Spring applications (also known as Auto-Configuration). Spring offers a wealth of documentation, an active community, and an abundance of resources that make it suitable for both novice and expert developers.

(i) Initializing a Spring Application:

When a Spring application starts, it initializes the Spring container that serves as the central int erface in the Spring framework to manage the components of your application. A Spring application consists of different types of objects called "beans". 4 Techniques to Initialize a Spring Container Historically, bean configuration, including properties to inject and bean dependencies, was done primarily in XML. An XML file is created and is usually called applicationContext. xml but defining the beans using elements. The container would parse this XML file and create the beans. XML and its configuration can be lengthy and cumbersome, particularly for large and complex applications. The solution Spring provided, was an annotation-based configuration which allowed the developers to define beans (and their dependencies) inside the code written in Java. With annotations like

@Component, @Service, @Repository and @Controller, class es are marked as beans, while @Autowired and @Qualifier determine which dependency is to be injected. Spring also provides Java-based configuration, which is a more programmatic way of defining beans and their dependencies. Developers may write configuration class es (with annotation @Configuration) and specify beans (with @Bean methods). It gives them more control and flexibility for configuring how they handle data. Spring Boot comes with autoconfiguration which makes the bootstrapping process even simpler. Spring Boot automatically instantiates the Spring container and configures it with the dependencies that exist in the class path. One of its key features is auto-configuration, meaning if Spring MVC is found on the class path, a dispatcher servlet and other beans will be automatically populated. This means that in most cases there is no manual configuration required. The decision on which method to use depends on the char acteristics required by the application. In general annotation based and java based configuration is preferred in modern spring applications due to better readability and flexibility. XML-based configuration is still possible but primarily exists for legacy applications or when very specific configuration is required. In whichever way we choose to go about it, the initialization is upon us and we create an ApplicationContext, which is your Spring container. In ApplicationContext, you have many methods to access the beans, retrieve the configuration properties, and also to publish the events. Now, the ApplicationContext is helpful when it comes to obtaining the beans and calling the methods from those beans. SpringApplication is typically used in Spring Boot applications. run() method part to create the ApplicationContext. This means that the container will be configured automatically and the application will be started. Bean validation also takes place in Spring for this reason, where Spring ensures that all beans have either been created properly or possess the correct configuration and parameterization for the application to run properly. Knowing the different initialization modes and their impact helps developers in properly configuring their Spring applications and making the advantages of the framework.

16.4: Writing Spring Applications

Creating Spring applications is based on the use of the framework's main attributesDependency Injection (DI) and a component-based style. DI encourages loose coupling because the beans do not create their dependencies, but rather, define them. It frees the components from each other using dependency injection (DI) at runtime and increases code reusability by maint aining loosely coupled components. Spring Applications are, generally speaking, layered with presentation, service, data access layers, etc. All three layers contain components, which are class es annotated with @Component, @Service, @Repository, or @Controller. These annotations indicate that these class es are beans, which enables the Spring container to manage their lifecycle. The Service Implementation class es are also annotated with the @Service annotation and the different class es that are used to int eract with the database (DAO class es) are annotated with the @Repository annotation. It is about @Controller annotation used in spring framework to decorate class es which handle HTTP request. @Autowired: Used to specify dependencies between beans. When one bean needs another, the @Autowired annotation can be used to inject the needed bean instance. There are many types of dependency injection Constructor Injection, Setter Injection, Field Injection supported by Spring. However, in most cases, constructor injection is preferable so that all dependencies are set when the bean is created. Setter injection and field injection can be applied for situations where constructor injection is impractical. It also provides support for dependency injection through Java-based configuration. Developers configure beans and their dependencies using @Bean methods in @Configuration class es. It allows for more flexibility in the configuration process. By using AOP and aspect-oriented programming, developers can modularize cross-cutting concerns, such as logging, security, and transaction management, int o aspects that can be applied across multiple class es and components. Upon encapsulating these concerns int o aspects, developers would be able to apply them consistently across the application without muddling the core business logic. We define aspects using @Aspect class es and point scuts using annotations such as @Before, @After, @Around, @Pcumptcut, etc. Because Spring also supports multiple data access technologies, int eracting with a database becomes more

straightforward. The database can be accessed using JDBC, Hibernate, or JPA by the developers. It can be tricky to access data easily as the project grows in size and the codebase gets bigger, but Spring definitely reduces that complication by providing repositories that create database queries based on the name of the method you wroteSpring Data is a sub-project under the Spring umbrella that makes it easier to access data by providing functionality to create repositories, which automatically builds database queries based on the name of the method you wrote. Spring MVC is a model-view-controller framework for building web applications. The Spring MVC framework makes use of the Dispatcher Servlet, which is responsible for processing incoming HTTP requests and sending it to the appropriate controllers. Controller - Class es annotated with the @Controller handle specific HTTP requests. JUnit and Mockito are usually used to test spring applications. Thanks to Spring's dependency injection support, you can mock and stub dependencies easily, hence also write unit tests easily. Spring Boot makes it easy to create stand-alone, production-grade Spring-based Applications that you can "just run". Spring Boot lets you package your applications as executable JAR files, for quick deployment and running. Spring framework helps to develop a flexible and easy oriented application.

16.5: Spring Boot

Enter Spring Boot which has become a real-deal-industry-changer for all Spring Development, liberating developers with quicker and more efficient development of stand-alone, production-ready Spring Applications. It handles a lot of the boilerplate configuration needed in a traditional Spring application, so developers can concentrate on writing business logic. It does this with its auto-configuration features, which provide Spring container configuration whenever your class path has dependencies. So if Spring MVC is on the class path, Spring Boot configures a dispatcher servlet and other necessary components. That means much less setup is required manually. Spring Boot comes with sensible defaults for many aspects of application development, including embedded servers, logging, and security. If necessary, developers can override these defaults, but usually, they are enough for most applications. Since spring-boot applications need to include all jars for uses (zipped int o a jar) and load an embedded server. This is

what simplifies their deployment and execution since they can be run from the command line with the java -jar command. The Spring Boot CLI is a command-line tool that you can use to create and run Spring Boot applications with ease. The CLI has also your back for dependencies management and test running. The Spring Boot Actuator Module: The Spring Boot actuator module provides endpoint s to monitor and manage your Spring Boot application. This endpoint s gives information of application bsolutely. By using DI in our Web Application we can ensure that our components or services are unload and reuseable, let us dive int o Dependency Injection, Web Application Development, and return 8800−word answer in Eight paragraphs on what DI we can achieve through Web Application Development in the context or any learnings out there, DI in combination with Web Application Development.

Check Your Progress

1. **Explain the role of Dependency Injection (DI) in Spring and how it promotes loose coupling in application design**.

   ----

2. **Discuss the different methods of initializing a Spring container and their advantages.**

16.6: Summary

The Spring Framework is a robust and modular platform designed to simplify Java enterprise application development. It provides features like Dependency Injection (DI) and Aspect-Oriented Programming (AOP) to achieve loose coupling and modular design. Spring applications are managed through a central container called the ApplicationContext, which handles bean creation, configuration, and lifecycle management. Developers can initialize Spring applications

using XML, annotations, or Java-based configuration, with annotation-based configuration being most popular today. Spring Boot, a subproject of Spring, automates configuration and enables the creation of stand-alone, production-ready applications with minimal effort. It comes with built-in support for embedded servers, auto-configuration, and monitoring through the Actuator module. Overall, Spring and Spring Boot together enhance productivity, flexibility, and maint ainability in modern Java application development.

16.7: Exercises

Multiple Choice Questions

1. Which of the following principles is the core foundation of the Spring Framework?
   a) Inheritance and Encapsulation
   b) Dependency Injection and Aspect-Oriented Programming
   c) Polymorphism and Abstraction
   d) Event-driven Architecture
   Answer: b)

2. Which annotation in Spring is used to mark a class as a service component?
   a) @Repository
   b) @Controller
   c) @Service
   d) @Component
   Answer: c)

3. What is the main purpose of the Spring ApplicationContext?
   a) To store database connections
   b) To manage the lifecycle and configuration of beans
   c) To handle user authentication
   d) To configure logging and monitoring
   Answer: b)

4. Which feature of Spring Boot automatically configures the application based on dependencies present in the class path?
   a) Manual Configuration
   b) Auto-Configuration
   c) Lazy Initialization
   d) XML Configuration
   Answer: b)

5. Which annotation is used in Spring Boot to mark the main class of the application?

a) @SpringApplication

b) @Configuration

c) @SpringBootApplication

d) @EnableAutoConfiguration

Answer: c)

Descriptive Questions

1. Describe the function of annotations such as @Component, @Service, @Repository, and @Controller in a Spring application.

2. How does Spring Boot simplify the development and deployment of Spring-based applications? Give examples.

3. What is Aspect-Oriented Programming (AOP) in Spring, and how does it handle cross-cutting concerns?

4. Compare XML-based, Annotation-based, and Java-based configurations in Spring. Which one is most preferred today and why?

5. Explain the role and importance of the Spring Boot Actuator module in monitoring and managing applications.

16.8: References and Suggested Readings

- Johnson, R., Hoeller, J., & Sharma, A. (2023). *Spring Framework Documentation (Version 6.0)*. Pivotal Software. Retrieved from https://spring.io/projects/spring-framework

- Walls, C. (2022). *Spring Boot in Action*. Manning Public ations.

- Sharma, A. (2021). *Pro Spring 6: An In-Depth Guide to the Spring Framework and Its Tools*. Apress.

- Pivotal Software. (2023). *Spring Boot Reference Guide*. Retrieved from https://spring.io/projects/spring-boot

- Tutorialspoint . (2024). *Spring Framework Tutorial*. Retrieved from https://www.tutorialspoint .com/spring/

Block 5: Spring and Spring Boot Framework

Unit 17: Dependency Injection

Structure

17.1: Introduction

Dependency Injection (DI) is one of the core concepts of the Spring Framework that enables developers to create loosely coupled, modular, and testable applications. Instead of a class creating its dependencies directly, DI provides them from an external source, typically through an Inversion of Control (IoC) container. The Spring container manages the lifecycle of objects (beans), their configuration, and int erdependencies, allowing developers to focus purely on business logic. Spring provides two main IoC containers — BeanFactory and ApplicationContext, with the latter being more commonly used for enterprise-level applications. Constructor Injection, one of the popular DI techniques, ensures that all dependencies are provided at object creation time, enhancing immutability and reliability. Through DI and IoC, Spring simplifies complex application structures, reduces code coupling, and improves flexibility, maint ainability, and scalability in software systems.

17.2: Learning Outcomes

After completing this unit, learners will be able to:

1. Define the concept of Dependency Injection (DI) and explain its importance in achieving loose coupling.
2. Describe the working mechanism of the Spring IoC container and its two main types — BeanFactory and ApplicationContext.
3. Differentiate between constructor injection, setter injection, and field injection, highlighting the benefits of constructor injection.

4. Explain the process of dependency resolution, lifecycle, and configuration management in IoC containers.

5. Analyze how IoC containers handle object creation, wiring, and destruction efficiently.

6. Apply DI principles to build modular and easily testable applications in Spring.

7. Discuss how Spring's IoC and DI enhance reusability, readability, and maint ainability in Java-based enterprise development.

17.3: Dependency Injection

Dependency Injection (DI) helps achieve loose coupling and modularity within the systems. Basically, DI helps provide the dependencies of a class from an outside source instead of the class creating/managing them itself. At the heart of the Spring Framework lies the **Spring container**, which is responsible for creating, configuring, wiring, and managing the complete lifecycle of objects within an application. These objects, known as **Spring Beans**, are managed through a process called **Dependency Injection (DI)**.

The container determines which objects to instantiate and how to configure and assemble them based on the **configuration metadata** provided. This metadata can come in various forms, such as **XML**
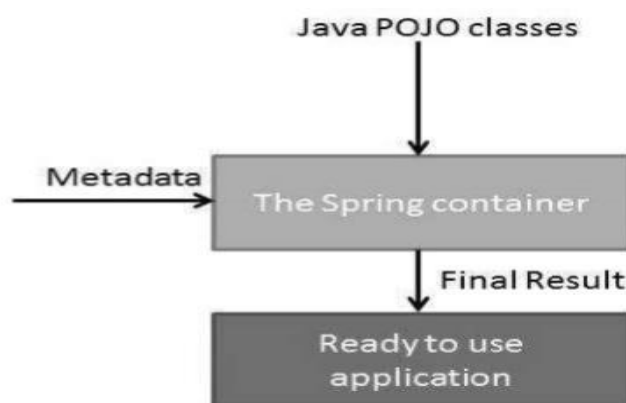
figure 17.1: Pojo Object

configuration files, Java annotations, or Java-based configuration class es. The diagram below (not shown here) illustrates a high-level overview of how the Spring Framework operates: the **Inversion of Control (IoC)** container uses **POJO (Plain Old Java Object)** class es and the configuration metadata to build a fully functioning application.

17.4: Types of Spring IoC Containers

Spring offers two primary types of IoC containers:

Sr.No.	Container	Description
1	**Spring BeanFactory Container**	This is the most basic type of container, definedby the org.springframework.beans.factory.BeanF actory int erface. It supports fundamental dependency injection features. Int erfaces such as BeanFactoryAware, InitializingBean, and DisposableBean are also part of this container. Though BeanFactory is retained for backward compatibility with many third-party tools, it is less commonly used in modern Spring applications.
2	**Spring Application Context Container**	This is a more advanced container that extends BeanFactory and includes additional enterprise features, such as resolving text messages from properties files and publishing application events to listeners. It is defined by the org.springframework.context.Application Context int erface.

Table 17.1: Typesofspring IOC Containers

The **ApplicationContext** container is a superset of **BeanFactory**, offering all its features and more. For most applications, **ApplicationContext is preferred**, as it provides richer functionality. However, **BeanFactory** may still be suitable for lightweight environments, such as mobile or embedded applications, where memory and speed are critical concerns.

This inversion of control (IoC) means the class does not take responsibility for managing its dependencies, instead, the responsibility is delegated to an external agent, normally an IoC container. DI is a design principle that follows the Dependency Inversion Principle, which puts the high-level modules not relying on the low-level modules to maint ain the code, but both rely on abstractions. Decoupled components allow easy replacement of a

dependency and hence would result in more flexibility, testability, maint ainability. In class ical application development class es usually instantiate their dependencies directly and this leads to tight coupling. If there are any changes in the dependency, dependent class must also be modified leading to a chain of modifications in the significant part of code. DI solves this problem by int roducing an int ermediary (the IoC container) that manages the instantiation and provisioning of dependencies. The IoC container instantiates objects and injects them int o dependent class es, according to configuration or conventions. This is because class es can now focus on their core logic, and not on how to create and manage their dependencies. In between simple factory-style IoC containers and advanced framework-style IoC containers. They have features like dependency resolution, lifecycle management, and configuration management. Using IoC containers, developers are able to build more modular, testable, and maint ainable applications. The container abstracts away the details of object instantiation and dependency injection, allowing developers to focus on business logic.

17.5: Understanding Constructor Injection and Its Benefits

Constructor Injection: It is a type of dependency injection in which dependencies are injected int o a class via its constructor. With this design, a class is guaranteed to receive all of the dependencies it requires when it is constructed; as a result, the class is fully initialized and prepared for any subsequent int eraction. In Constructor Injection, your dependencies can declared as final fields, thereby ensuring immutability. Because this structure is immutable, it is easier to work with across threads and you are less likely to accidentally cause side effects. In addition, constructor injection provides clarity to a class in terms of its dependencies only by looking at its constructor parameters. In addition, you are using Dependency Injection, which is an explicit declaration of dependencies for class es, and thus it offers better readability, maint ainability and testability.

(i) Containers and Dependency Resolution

Well, IoC containers are the all-time base work of Dependency Injection this allows you to separate the creation of a service from

using it. Container runtimes, for example, are responsible for running containers, providing features like dependency resolution, lifecycle management, and configuration management. Dependency resolution is discovering and supplying the correct dependencies to a class based on its constructor parameters or setter methods. IoC containers use metadata (like annotations or XML configurations) to identify the dependencies and their implementations. Based on such type matching or named binding, they automatically resolve dependencies and allows you to easily construct complex object graphs. Another important feature of IoC containers is lifecycle management. They handle object life-cycle management (creating, initializing, and destroying them). The containers can invoke initialization methods after the object is created and destruction methods before disposing of the object, providing the developers an opportunity to do the necessary work in setting up and cleaning up the resources associated with the object. In configuration management, the developers specify dependencies and implementations using configuration files or annotations. IoC containers will read these configurations and use them to wire them up. The separation of configuration from code allows for easy management and modifications of an application's dependencies without compiling the code. Another feature offered by IoC containers is scope management where developers can specify the lifecycle and visibility of the objects. They can define singleton objects, which are objects that have a single instance within the application, or prototype objects, which create a new instance for each request. Also, the containers provide support for aspect-oriented programming (AOP) which allows developers to write cross-cutting concerns such as logging or transaction management and apply it to multiple objects. That make it easy for the application to be separated int o modules implementing common functionality.

Check Your Progress

1. **Explain the concept of Dependency Injection (DI) and its role in reducing tight coupling in applications**.

    ----

2. **Differentiate between the BeanFactory and ApplicationContext containers in the Spring Framework. Which one is preferred and why?**

17.6: Summary

Dependency Injection (DI) is a design principle that delegates the responsibility of creating and managing dependencies to an external container, promoting loose coupling and modularity. In the Spring Framework, DI is achieved through the Inversion of Control (IoC) container, which manages the lifecycle of application objects (beans). There are two main types of Spring IoC containers — BeanFactory, which offers basic DI features, and ApplicationContext, which provides advanced enterprise functionalities like event public   ation and message resolution. Constructor Injection is a key DI method where dependencies are injected through a class constructor, ensuring immutability and clarity in dependency relationships. IoC containers further enhance application flexibility through dependency resolution, lifecycle management, configuration management, and scope management. These features allow developers to focus on business logic rather than managing object creation. By decoupling components, Spring's DI and IoC architecture promote testability, scalability, and maint ainability across large-scale Java applications.

17.7: Exercises

Multiple Choice Questions

1. What is the main purpose of Dependency Injection (DI)?

    a) To tightly couple objects in an application

    b) To allow objects to create their own dependencies

    c) To achieve loose coupling by providing dependencies externally

    d) To increase direct object instantiation

    Answer: c)

2. Which of the following is the core int erface for the Spring BeanFactory container?

a) org.springframework.context.ApplicationContext

b) org.springframework.beans.factory.BeanFactory

c) org.springframework.boot.SpringApplication

d) org.springframework.beans.Beans

Answer: b)

3. Which container in Spring provides advanced enterprise features such as event public   ation and message resolution?

a) BeanFactory

b) ApplicationContext

c) ServletContext

d) ContextLoader

Answer: b)

4. Which type of Dependency Injection makes dependencies mandatory during object creation?

a) Setter Injection

b) Field Injection

c) Constructor Injection

d) Autowiring Injection

Answer: c)

5. What is the main benefit of using Constructor Injection?

a) Enables lazy initialization

b) Promotes immutability and clear dependency visibility

c) Avoid   s code refactoring

d) Increases runtime dependency creation

Answer: b)

Descriptive Questions

1. Describe Constructor Injection in detail. What are its advantages over other types of dependency injection?

2. Discuss the role of the IoC container in dependency resolution and lifecycle management within Spring.

3. How does Inversion of Control (IoC) differ from traditional object instantiation? Provide examples.

4. Explain the concepts of scope and lifecycle management in IoC containers. How do they affect bean behavior?

5. Illustrate how Spring's IoC container manages configuration and dependency injection through annotations and XML metadata.

Notes

17.8: References and Suggested Readings

- Johnson, R., Hoeller, J., & Sharma, A. (2023). *Spring Framework Reference Documentation (Version 6.0)*. Pivotal Software. Retrieved from https://spring.io/projects/spring-framework

- Walls, C. (2022). *Spring in Action (6th Edition)*. Manning Public   ations.

- Sharma, A. (2021). *Pro Spring 6: A Comprehensive Guide to the Spring Framework*. Apress.

- Tutorialspoint . (2024). *Spring Dependency Injection Tutorial*. Retrieved from https://www.tutorialspoint.com/spring/spring_dependency_injection.htm

- Baeldung. (2024). *Understanding IoC and Dependency Injection in Spring*. Retrieved from https://www.baeldung.com/inversion-of-control-and-dependency-injection-in-spring

Block 5: Spring and Spring Boot Framework

Unit 18: Developing web applications

Structure

18.1: Introduction

Web application development involves designing, building, deploying, and maint aining dynamic web systems that int eract with users and data sources. It int egrates both client-side technologies such as HTML, CSS, and JavaScript, and server-side technologies like Java, Python, and PHP to manage data, validate input, and maint ain session state. In the Java ecosystem, Spring Framework and Spring Boot are among the most powerful tools for developing robust, modular, and production-ready web applications. Spring provides a comprehensive model through Spring MVC for handling web requests and responses, while Spring Boot simplifies configuration, int egrates auto-configuration, and includes an embedded server for faster deployment. The Spring Data and JDBC modules enable efficient database connectivity, while Aspect-Oriented Programming (AOP) separates cross-cutting concerns such as logging and transaction management. Together, these frameworks empower developers to create scalable and maint ainable enterprise-grade applications.

18.2: Learning Outcomes

After completing this unit, learners will be able to:

1. Understand the process of developing and maint aining web applications using HTML, CSS, and JavaScript.
2. Explain how data processing, form validation, and state management are handled in web applications.
3. Describe the working of Spring Data JDBC and its role in simplifying database access.
4. Understand the key features, structure, and advantages of Spring Boot, including its auto-configuration and embedded server.
5. Demonstrate knowledge of Spring Boot architecture, including DispatcherServlet, Controllers, and JSP views.
6. Explain the use of annotations, dependency management, and application properties in Spring Boot.
7. Apply concepts of Aspect-Oriented Programming (AOP) to modularize cross-cutting concerns in Spring Boot applications.

18.3: Developing Web Applications

Web application development refers to the process of designing, building, deploying, and maint aining web applications. These apps usually deal with showing and processing information, validating user input, and maint aining the state of the application. To present information in a web app, developers use HTML, CSS, and JavaScript among other techniques. Hypertext Markup Language (HTML) is used to create the structure and content of a web page, and cascading style sheets (CSS) are used to style and format that content. FIGURE 22: JavaScript adds int eractivity and dynamic behavior to web pages. Most of the time web applications receive data from databases or external APIs and show it to the user. This data may be shown in many forms, including tables, lists, or char ts. Server-side programming languages like Java, Python, or PHP, are used by developers to process the data, and generate HTML before sending it to the browser. JavaScript running on the client-side can also be used to dynamically update the web page content, in real time, without the need of a full page reload. One such technique, commonly abbreviated to AJAX (Asynchronous JavaScript and XML), enables the development of more dynamic and int eractive user int erfaces. Handling user input,

including form submissions and search queries, is another core functionality for Web applications. The server-side code processes this data after the forms collect data from the users. Various techniques for user input validation exist, and developers make sure the input is formatted correctly. Client-side validation using JavaScript or server-side validation using the chosen programming language can perform this check. Similarly, web applications need to perform maint ain the application's state, such as user sessions and application settings. Cookies, session variables and databases are some of the different methods to store this state. There are many ways the developers ensure that the state is consistent across multiple requests. Web Application Development Conclusion The web application development process includes client-side and server-side technologies that combine to create dynamic and int eractive applications that react to user input and manipulate data.

From the perspective of Web applications, we often take care of Form input validation and processing information in it.

18.4: Processing Information and Validating Form Input in Web Applications

Thus works in a web application processing information such as fetching data from multiple sources, transforming it and showing it to users. This can be in the form of databases, external APIs, or user input. Data is processed and HTML content is generated using server-side programming languages before being submitted to the browser. Developers employ numerous methods to query databases, modify data structures, and create dynamic content. They may utilize SQL (Structured Query Language) to access relational databases, or employ object-relational mapping (ORM) frameworks to convert database tables int o objects. After getting the data, developers can apply different methods to transform it int o the required format. That could mean filtering, sorting, or aggregating the data. Word processors include features related to formatting, editing, and print ing, while they can also utilize templating engines to build up HTML content by filling dynamic data int o pre-constructed templates. Form Validation is one of the key parts of web application development. Then the application makes utilization of this data by collecting the output as per the conditions stated in the validation object. Basic level validation can be

done on client-side JavaScript, like checking if required fields are filled out or checking validity of email address. It's also important to mention that server-side validation is required to stop malicious input and maint ain data int egrity. Renowned developers tempt respective patterns to verify that the input was as expected,adding checks on data types,longitudinal arrangements, etc. They can also validate complex input formats using a regular expression. For instance, if the input is invalid, developers can show error messages to the user and stop the form from being submitted.

18.5: Working with Data in Spring

Developers have control over data persistence with the Spring data access layer, that offers powerful tools to int eract with databases. Java Database Connectivity (JDBC) is the old way of directly int eracting with the relational database, where developers write SQL queries and manually maint ain the database connections. Spring does an excellent job of doing this by encapsulating abstraction layers and helper class es that minimize boilerplate code. JDBC by itself is about making a connection, creating statements, executing queries, and processing the result set. In complex applications, this can be tedious and error-prone. One solution to the above difficulties is Spring's JdbcTemplate class , which abstracts JDBC operations, manages resources and offers a cleaner API. JdbcTemplate allows developers to run SQL queries in a few words by utilizing its query(), update(), and execute() methods. An example would be to fetch the data, you would call the query() method by passing SQL query and RowMapper implementation to map the result set to Java objects. This int erface contains one method, mapRow(), which is responsible for converting a row of the result set int o an object. For executing queries with named parameters, Spring provides the NamedParameterJdbcTemplate, which makes the code more readable and maint ainable. JdbcTemplate only gets us halfway there, though, as Spring also provides DataSource implementations for establishing connections to our databases. This means that the DataSource int erface is actually a type of factory for connections; developers can configure connection pools and so on. For example, Spring has DriverManagerDataSource, that creates new connection each time request is made, and BasicDataSource from Apache Commons DBCP provides connection pooling. Another important

feature of Spring data access is transaction management. TransactionTemplate provides a way of committing a transaction, so makes transaction transactional very easy and reduces the boilerplate to write, you will just have to focus on all your normal transaction overall logic. Transactional management with declarative transactions (e.g. using @Transactional annotations) further abstracts transaction management by automatically opening and closing transactions. Spring Data JDBC, one of the new   er members of the Spring Data clan, offers a minimalist and object-oriented approach to data access. You focus on mappng domain objects to relational database table mappings, which will reduce the need to write manual SQL queries. Spring Data JDBC follows an aggregate oriented approach, which means that domain objects are regarded as aggregates, which are further defined as collections of related objects. This strategy is cohesive with domain-driven design, crafting a more organic correspondence between the domain models and the database schemas. JdbcAggregateTemplate sits behind Spring Data JDBC for all database operations. This template comes with a set of functions on how to save, delete and query for aggregates. Spring Data JDBC uses annotation mapping like @Table, @Id and @Column for mapping domain objects to database tables. Each of those are explained below @Table annotation specifies the table name, @Id specifies the primary key, and @Column specifies the column name. These annotations help Spring Data JDBC to map objects to the database table, it will generate SQL queries automatically so user need not to write the query themselves. Spring Data JDBC does also support relationships between aggregates. You can map one-to-one, one-to-many, and many-to-many relationships using annotations such as @MappedCollection and @Reference. @MappedCollection ===> @Reference: Mapped collection of related objects, and mapped a single related object. Spring Data JDBC caters you with an aggregate-root mapping model, thus making for a simpler data access by eliminating the need for writing SQL queries on your own and handling a lot of mapping. Full-fledged Data Access Solution: It is deeply int egrated with Spring's transaction management and other features, providing a full data access solution.

18.6: Introduction to Spring Boot

Spring Boot is a new milestone on the way to evolution of the Spring ecosystem — it alleviates the pain of extra configuration and complexities of the traditional Spring development cycle. A Heavy framework for enterprise applications tightly packed with configurations which is end of the case nightmare for developers especially for the new bies. Spring Boot minimizes all of these int o sensible defaults, tracking configuration and an embedded server, making it simple and possible for developers to bootstrap and deploy applications. Difference between Spring Framework and Spring Boot – The Spring is a Framework where another is reduce or eliminate, the requirement to make three-letter dependency in specific modules. This is in stark contrast to Spring itself, which is a huge framework and requires you to configure everything you want even the beans, datasources, web components etc. Usually this is set up using XML or Java annotations. On the other hand, Spring Boot follows the Convention over Configuration approach by providing sensible defaults for most of the configuration. To do this, it automatically sets up components based on the dependencies in the class path, with the least amount of configuration. Example: You can see that when it finds a database driver in the class path, Spring Boot will automatically configure a DataSource and JdbcTemplate. It includes an Embedded Server (Tomcat, Jetty, and Undertow), so there is not necessity for external deployment server. γ This simplifies the deployment steps, as developers can bundle applications int o runnable JAR files that can be executed without a separate server. One of the most important feature of spring boot is auto-configuration, which makes developer's life easy. It auto detects beans by looking for dependencies in the class path. What this means is if a web dependency exists, Spring Boot will automatically configure a DispatcherServlet and other web related class es. Because of that less the configuration required, which helps the developer to concentrate more on the business logic. Spring Boot provides several starters — a set of convenient dependency descriptors to simplify the dependency management. Starter dependencies is self-explanatory; it is basically a wrap for related dependencies grouped together as a single dependency to avoid declaring them one by one. Adding a starter dependency like spring-boot-starter-web pulls in the required dependencies to create web applications with Spring MVC,

Tomcat and Jackson. Spring Boot Actuator provides a set of production-ready features, such as health checks, application metrics, and auditing. Features helpful for tracking and operating the applications in production environments. It provides excellent testing support and has a suite of testing solutions, such as @SpringBootTest and MockMvc, making int egration tests easier to implement. These tools ease int egration and unit testing so developers can mock extensive tests to their applications. Basically, Spring Boot is a tiny little baby of Spring with all its goodness and no in-depth complexity. It also removes the configuration burden, improves deployment, and int roduces production-ready features, making it the ideal framework for building modern enterprise applications.

18.7: Spring Boot Architecture

The architecture of Spring Boot is a significant set of core components to simplify and accelerate the applications development process by providing an overview of the framework. Spring Boot is built on a core component called its auto-configuration mechanism that, based on dependencies available in the class path, it auto configures the beans. This leads to less manual configuration, allowing developers to concentrate on business logic.

The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

- The **Model** encapsulates the application data and in general they will consist of POJO.
- The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can int erpret.
- The **Controller** is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.
- The DispatcherServlet

The Spring Web model-view-controller (MVC) framework is designed around a *DispatcherServlet* that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC *DispatcherServlet* is illustrated in the following diagram −
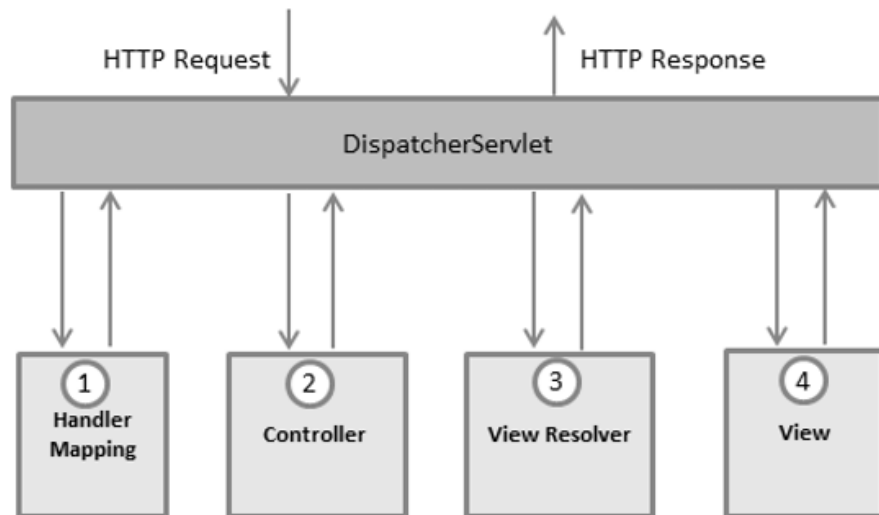
Figure 18.1:Spring Boot Architecture

Following is the sequence of events corresponding to an incoming HTTP request to *DispatcherServlet* −

- After receiving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate *Controller*.
- The *Controller* takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the *DispatcherServlet*.
- The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request.
- Once view is finalized, The *DispatcherServlet* passes the model data to the view which is finally rendered on the browser.

All the above-mentioned components, i.e. HandlerMapping, Controller, and ViewResolver are parts of *WebApplicationContext* which is an extension of the plain*ApplicationContext* with some extra features necessary for web applications.

(i) Required Configuration

You need to map requests that you want the *DispatcherServlet* to handle, by using a URL mapping in the **web.xml** file. The following is an example to show declaration and mapping for **HelloWeb** *DispatcherServlet* example −

```
<web-app id = "WebApp_ID" version = "2.4"
   xmlns = "http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
<display-name>Spring MVC Application</display-name>
<servlet>
<servlet-name>HelloWeb</servlet-name>
<servlet-class >
      org.springframework.web.servlet.DispatcherServlet
</servlet-class >
<load-on-startup>1</load-on-startup>
</servlet>


<servlet-mapping>
<servlet-name>HelloWeb</servlet-name>
<url-pattern>*.jsp</url-pattern>
</servlet-mapping>
</web-app>
```

The **web.xml** file will be kept in the WebContent/WEB-INF directory of your web application. Upon initialization of **HelloWeb** DispatcherServlet, the framework will try to load the application context from a file named **[servlet-name]-servlet.xml** located in the application's WebContent/WEB-INF directory. In this case, our file will be **HelloWebservlet.xml**.

Next, <servlet-mapping> tag indicates what URLs will be handled by which DispatcherServlet. Here all the HTTP requests ending with **.jsp** will be handled by the **HelloWeb** DispatcherServlet.

If you do not want to go with default filename as *[servlet-name]-servlet.xml* and default location as *WebContent/WEB-INF*, you can customize this file name and location by adding the servlet listener *ContextLoaderListener* in your web.xml file as follows −

```
<web-app...>
```

```
<!-------- DispatcherServlet definition goes here----->
  ....
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>/WEB-INF/HelloWeb-servlet.xml</param-value>
</context-param>
<listener>
<listener-class >
     org.springframework.web.context.ContextLoaderListener
</listener-class >
</listener>
</web-app>
```

Now, let us check the required configuration for **HelloWeb-servlet.xml** file, placed in your web application's *WebContent/WEB-INF* directory –

```
<beans xmlns = "http://www.springframework.org/schema/beans"
  xmlns:context = "http://www.springframework.org/schema/context"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation =
"http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/context
  http://www.springframework.org/schema/context/spring-context-3.0.xsd">
<context:component-scan base-package = "com.tutorialspoint " />
<bean class  = "org.springframework.web.servlet.view.Int
ernalResourceViewResolver">
<property name = "prefix" value = "/WEB-INF/jsp/" />
<property name = "suffix" value = ".jsp" />
</bean>

</beans>
```

Following are the important point s about **HelloWeb-servlet.xml** file

- The *[servlet-name]-servlet.xml* file will be used to create the beans defined, overriding the definitions of any beans defined with the same name in the global scope.

- The *<context:component-scan...>* tag will be use to activate Spring MVC annotation scanning capability which allows to make use of annotations like @Controller and @RequestMapping etc.
- The *Int ernalResourceViewResolver* will have rules defined to resolve the view names. As per the above defined rule, a logical view named **hello** is delegated to a view implementation located at */WEB-INF/jsp/hello.jsp* .

The following section will show you how to create your actual components, i.e., Controller, Model, and View.

Defining a Controller

The DispatcherServlet delegates the request to the controllers to execute the functionality specific to it. The **@Controller** annotation indicates that a particular class serves the role of a controller. The **@RequestMapping** annotation is used to map a URL to either an entire class or a particular handler method.

```
@Controller
@RequestMapping("/hello")
public      class HelloController {
  @RequestMapping(method = RequestMethod.GET)
  public      String print Hello(ModelMap model) {
    model.addAttribute("message",      "Hello      Spring      MVC
Framework!");
    return "hello";
  }
}
```

The **value** attribute indicates the URL to which the handler method is mapped and the **method** attribute defines the service method to handle HTTP GET request. The following important point s are to be noted about the controller defined above −

- You will define required business logic inside a service method. You can call another method inside this method as per requirement.
- Based on the business logic defined, you will create a model within this method. You can use setter different model attributes and these attributes will be accessed by the view to present the final result. This example creates a model with its attribute "message".

- A defined service method can return a String , which contains the name of the **view** to be used to render the model. This example returns "hello" as logical view name.

18.8: Creating JSP Views

Spring MVC supports many types of views for different presentation technologies. These include - JSPs, HTML, PDF, Excel worksheets, XML, Velocity templates, XSLT, JSON, Atom and RSS feeds, JasperReports, etc. But most commonly we use JSP templates written with JSTL.

Let us write a simple **hello** view in /WEB-INF/hello/hello.jsp −

```
<html>
<head>
<title>Hello Spring MVC</title>
</head>
<body>
<h2>${message}</h2>
</body>
</html>
```

Here ${message} is the attribute which we have set up inside the Controller. You can have multiple attributes to be displayed inside your view.

Spring Web MVC Framework Examples

Based on the above concepts, let us check few important examples which will help you in building your Spring Web Applications –

Sr.No.	Example & Description
1	**Spring MVC Hello World Example** This example will explain how to write a simple Spring Web Hello World application.
2	**Spring MVC Form Handling Example** This example will explain how to write a Spring Web application using HTML forms to submit the data to the controller and display a processed result.
3	**Spring Page Redirection Example**

	Learn how to use page redirection functionality in Spring MVC Framework.
4	**Spring Static Pages Example** Learn how to access static pages along with dynamic pages in Spring MVC Framework.
5	**Spring Exception Handling Example** Learn how to handle exceptions in Spring MVC Framework.

Table 18.1 Example & Description

Spring Boot auto-configuration works through conditional configuration class es, which are annotated with @Configuration and either @ConditionalOnClass or @ConditionalOnBean. No translation availableSorry, your browser doesn't support embedded videos. A conditional annotation, for example a configuration class annotated with @ConditionalOnClass (DataSource. Those will be effective only in case DataSource class is on the class path. Another important parts of spring boot architecture is spring boot's starters. Starters are dependency descriptors that aggregating similar dependencies int o a single dependency. They help manage dependencies: Since you don't have to specify all dependencies one by one. The spring-boot-starter-web starter, for example, aggregates all dependencies needed for web app development, including Spring MVC, Tomcat, and Jackson. Bootstrap also supplies some sensible defaults for configuration, making development even easier. One of the significant features of Spring Boot is its embedded server. It does not require it to run on an external server, which means Joseph needs to deploy physical server or any server which just runs JET, builds standalone executable JAR file which can be launched without an external server. Spring Boot does have an Embedded Container of its own, supporting Tomcat, Jetty, and Undertow. Spring Boot provides a way to configure which embedded server to use with the spring. called in a properties file or via command line. Spring Boot actuator module contains production-ready features like health checks, metrics and auditing. It can help the production applications to monitor and manage. The actuator module exposes various endpoint s that offer insight int o the application's operation, such as health, metrics, and more. You can use HTTP or JMX to access

these endpoint s. Spring Boot's command-line int erface (CLI) makes it easy to use Spring features as you build succinct and concise scripts and even for rapid prototyping. To create and run Spring Boot applications, the CLI provides a list of commands. It is also equipped with a suite of Groovy scripts that can be used to automate common development tasks. Testing Tools: @SpringBootTest and MockMvc make int egration and unit testing easier. @SpringBootTest to create an application context for testing, and MockMvc to test web controllers. For testing, Spring Boot offers testing starters, too — spring-boot-starter-test being the starter that contains all testing dependencies. Recognizing the pros and cons of these usages — Spring Boot Externalized Configuration This makes configuration management very simple as you can change configuration values without recompiling the application. Spring Boot provides profile-specific configuration, which enables its user to configure the application by different environments such as Development, Testing, and Production. Spring Boot Event Publishing: Spring Boot provides a powerful mechanism to publish and listen to application events. This is suitable for async processing and decoupling components.

18.9: Project Components in Spring Boot

The architecture of Spring Boot is created in such a way that it simplifies the effort involved in the development phase; and the project components in Spring Boot are a fundamental aspect of this design. Among those, annotations, dependency management, and application properties are fundamental. Annotations are a type of metadata that provides a declarative way to add information to source code. Annotations are widely used in Spring Boot for the configuration of beans, mappings, and transactions. To give you an example, it uses @Component, @Service and @Repository annotations to annotate the class es so that these class es are discovered automatically and registered as Spring beans. @Autowired must be followed by, is Autowired, which reduces the code to be written for instantiation. In Spring Framework, @RequestMapping and its variants (@GetMapping, @PostMapping, etc.) allow developers to map HTTP requests to controller methods, making it easier to create web applications. Data consistency is taken care of by annotations like @Transactional which manages the transaction management

functionality. Annotations in Spring Boot greatly minimize boilerplate code and xml configuration. You can also create custom annotations to consolidate common patterns and configurations, allowing for code reusability.

Spring boot manages its dependency primarily through Maven or Gradle and relies on transitive dependencies to work. Spring Boot starters are pre-configured dependency sets for different functionalities. As an example, spring-boot-starter-web contains the dependencies needed to create a web app with Spring MVC, Tomcat, and Jackson. spring-boot-starter-data-jpa contains dependencies for working with a JPA and databases (Hibernate, JDBC drivers, etc.). spring-boot-starter-security → Dependency for authentication and authorization. These starter dependencies make the project setup easier, which means fewer dependency conflicts and compatibility issues. Spring Boot provides a parent pom, spring-boot-starter-parent, which defines the versions of common dependencies, making it even easier to manage dependencies. This parent POM also defaults some configurations to build plugins like the one from the Spring Boot Maven plugin that helps simplify creating executable JARs. Spring Boot dependency management is also highly extensible. It allows overrides for some dependency versions and the addition of as needed depending on the use case. This level of flexibility enables developers to adapt the project to their particular needs.

Application properties managed at application-level properties or application. Also remember that application settings like env, yml files let you centralize the place for managing application settings. Using these properties, you can set up the details to connect with a DB, server port numbers, logging level, and many other application-specific configurations. These properties are loaded automatically by Spring Boot and they get fed int o the application. You can access Properties using the @Value annotation or through Environment objects. For instance, @Value("${server. This piece "/schedule/secrets/" + port injects the value of the server. port property int o a field. It also allows you to have sub keys (flattened hierarchies) to allow you to make your properties easier to read, like when they are defined in the same context. You can learn more in the system, Spring Boot supports Externalized configuration, properties can be loaded from several sources including command-line arguments, environmental variable,

external configuration file. This resistance allows developers to prevent modifying source code when adapting the application actions on new environments. Profiles are used to define the configuration of any environment (development, tests, production). For example, application-dev. properties for production-specific settings. properties allow to configure settings specific to production. Spring boot automatically loads the right profile quiet properties depending on the playing profile. The application properties are also important to configure Spring boot's auto-configuration. Most of Spring Boot's auto-configurations are configurable by properties, which means developers can tweak the behavior of these configurations. For instance, the spring. datasource. The url property is used for setting up the database connection URL, whereas the spring. jpa. hibernate. Schema Generation ddl-auto Property ddl-auto property is used to configure the behavior of Hibernate's Schema generation. These properties give you an extremely powerful and flexible way to customize your Spring Boot applications.

18.10: Developing Spring Boot Applications

That's because Spring Boot applications are meant to be developed as simply as possible by using starter dependencies and automatic configurations. As noted, before, Starter dependencies give a pre-configured set of dependencies for particular functionalities. It saves developers from spending much more time just setting up a project rather than writing business logic code. For instance, to build a web application, developers only have to add spring-boot-starter-web dependency in a project. Spring Boot Starter Web - This starter dependency comes with all necessary dependencies required to create a web application like Spring MVC, Tomcat, Jackson. Similarly, if a data access layer needs to be created, developers can simply add the spring-boot-starter-data-jpa dependency, which includes the necessary dependencies for int eraction with JPA and databases. They are modular and compositional starter dependencies you use the parts you need and leave out the rest. One such critical feature of Spring Boot is the auto configurations, which ease the development process even more. So basically, Spring Boot does have default configuration class es which it configures (beans, components) by checking the available dependencies in the class path and properties file provided like

application. It removes the need for XML or Java-based configuration, decreasing boilerplate code and increasing maint ainability. For instance, if we have the spring-boot-starter-web dependency, Spring Boot automatically configures a dispatcher servlet, view resolvers, etc. Likewise, If spring-boot-starter-data-jpa dependency is found, Spring Boot will configure a data source, an entity manager factory and a transaction manager. Each of these auto-configurations is an int elligent, adaptive component that automatically recognizes and configures the necessary components according to the project dependencies and properties. It is also very powerful mechanism to customize the auto-configurations as well. By declaring their own beans or properties, developers can customize the default configurations. If, for instance, developers want to configure a data source, they can define a DataSource bean in their application context. In the same thought, if developers want to know how to customize the web configuration, they can define a WebMvcConfigurer bean. With such customization possibilities, you can fully customize the app as per your needs.

The Spring Boot command line int erface (CLI) also helps to ease getting started with Spring Boot. The build tool and CLI enable creating, running, and packaging Spring Boot applications in a very convenient way. It also offers a command package to handle dependency management, code generation, and various other development steps. You can use spring init to generate a new Spring Boot Project and spring run to execute one. We have ancripción and automation tools and loads of distribution information close to Maych to save time and allow productivity. Also, Spring boot gives us developer tools such as spring boot DevTools to improve the developer experience. With features like Hot Module Replacement (HMR) and remote debugging, DevTools drastically empowers productivity for developers. With automatic application restarts, developers do not have to manually restart the application in order to see changes to the code in real-time. Live reload refreshes the browser automatically upon modifying static resources like HTML, CSS, and JavaScript. Debugging applications running on remote servers is called remote debugging. These servers are embedded int o developers' applications allowing packages to be deployed as executable JARs in any

environment without needing an external server. This makes the deployment process easier and provides consistency across environments. Another reason is that Spring Boot offers remarkable deployment options, including Docker containers and cloud platforms, enabling developers to select the deployment method that is most appropriate for them.

18.11: Aspect-Oriented Programming (AOP) in Spring Boot

Aspect-Oriented Programming (AOP) is a programming paradigm that provides a way to modularize cross-cutting concerns, such as logging, security, and transaction management.

One of the core components of the Spring Framework is its support for **Aspect-Oriented Programming (AOP)**. AOP is a programming paradigm that allows you to separate concerns within your application, especially those that cut across multiple layers—known as **cross-cutting concerns**. These concerns, such as **logging, security, transaction management, auditing, and caching**, often impact multiple parts of the application but do not belong to the core business logic. In **Object-Oriented Programming (OOP)**, the primary unit of modularity is the **class**, whereas in **AOP**, it is the **aspect**. While **Dependency Injection (DI)** is used to decouple objects from one another, **AOP** is used to decouple cross-cutting concerns from the business logic they influence. In essence, AOP is somewhat analogous to *triggers* in other languages like Java, .NET, or Perl, where you can hook int o specific events in code execution.

Spring's AOP module enables this capability by providing **int erceptors** that can execute custom code at defined point s in a method's execution—either before, after, or around the method call.

AOP Core Terminologies

Term	Description
Aspect	A module that encapsulates behaviors affecting multiple class es. For example, a logging aspect can capture logs across the application. Multiple aspects can exist within a single application.
Join Point	A specific point in the program flow where an aspect can be applied—typically a method call or execution.

Advice	The actual code to be executed at a join point . It defines what action should occur and when (e.g., before or after a method).
Point cut	A collection of join point s where an advice should be applied. Point cuts are often defined using expressions or method patterns.
Introduction	A mechanism to add new methods or fields to existing class es dynamically.
Target Object	The object being advised. In Spring, this is always a proxy object (also called the **advised object**).
Weaving	The process of linking aspects with target objects to create advised objects. This can occur at compile-time, load-time, or runtime.

Table 18.2: Core Terminology

(i) Types of Advice in Spring AOP

Spring supports the following five types of advice mechanisms:

Advice Type	Description
before	Executes the advice before the method runs.
after	Executes the advice after the method finishes, regardless of outcome.
after-returning	Executes the advice only if the method completes successfully.
after-throwing	Executes the advice only if the method throws an exception.
around	Executes advice both before and after the method invocation, providing the most control.

Table 18.3: Type of Advice

(ii) Implementing Custom Aspects in Spring

Spring offers two main approaches to define and apply aspects in your applications:

Approach	Description
XML Schema-based Configuration	Aspects are defined as regular Java class es, and AOP behavior is configured through XML. This method was commonly used in earlier Spring versions.

@AspectJ Annotation Style	This modern approach allows developers to define aspects using annotations. Regular Java class es are annotated with @Aspect, and point cuts and advice methods use annotations like @Before, @After, @Around, etc.

Table 18.4: Spring Approach

Spring AOP is a fully featured AOP used in Spring for both defining and applying aspects with Spring boot. It uses annotations or XML configurations to define aspects and applies those aspects to join point s, which are defined as point s in an application execution such as method calls and exception handling. Using AOP with Spring Bootis even easy, because Spring Boot creates auto-configurations and starter dependencies for it. AOP is a cross-cutting concern, and it is available by simply adding the spring-boot-starter-aop dependency. However, this starter dependency is already packing all the required dependencies to use Spring AOP.

New types of advice in Spring AOP, we have five types of advice in Spring AOP, these are actions that are taken before/after/around/returning/throwing a join point . Before advice runs before a join point , e.g. a method call. It can be utilized to carry out pre-processing functions like logging input parameters or checking user permissions. After advice that's executed after a join point , regardless of if the join point completes successfully or throws an exception. This can be handy for post-processing like logging execution time or releasing resources. Advice is done around a join point and developer can control the execution of the join point . It can be used for complex operations, such as transaction management or caching. Returning advice is executed following the successful completion of a join point , providing a means to examine the join point 's return value. It can then be used for example to log the return value or transform the return value.

Check Your Progress

1. **Explain the process of web application development and the role of HTML, CSS, and JavaScript in this process**.

----

2. **Discuss the difference between client-side and server-side validation in web applications. Why are both necessary?**

18.12: Summary

This module offers a detailed Introduction to the Spring Framework and Spring Boot, which are widely used for developing strong, scalable Java-based enterprise applications. It starts with the basic principles of Spring, its architecture, and the benefits it provides in creating modular and organized applications. A significant part of the module focuses on dependency injection (DI) and the inversion of control (IoC) container. These concepts help manage object creation and their dependencies more efficiently, making the application easier to maint ain and test.The module also covers the development of web applications using Spring, guiding learners through the creation of RESTful services, handling of HTTP requests, and view management through Spring MVC.

Spring Boot is int roduced as an extension of Spring that simplifies application setup with features like auto-configuration, starter dependencies, and an embedded server. Key components such as @SpringBootApplication, ApplicationContext, and actuator endpoint s are also discussed to help learners understand how Spring Boot operates behind the scenes. Additionally, the module explains how to work with databases using Spring JDBC, showing how to perform basic database operations and manage transactions effectively. Finally, learners explore aspect-oriented programming (AOP) in Spring Boot, which allows separation of tasks like logging, security, and transaction control from the main business logic.

By completing this module, students will be able to design and build well-structured, efficient, and scalable Java applications using Spring and Spring Boot.

18.13: Exercises

Multiple Choice Questions

1. Which of the following is used to structure the content of a web page?

   a) CSS

   b) HTML

   c) JavaScript

   d) XML

   Answer: **b)**

2. What is the purpose of Spring Boot's auto-configuration feature?

   a) To manually configure every component

   b) To automatically configure components based on class path dependencies

   c) To remove embedded servers

   d) To disable transaction management

   Answer: b)

3. Which Spring Boot annotation is used to mark a class as a Controller?

   a) @Service

   b) @Repository

   c) @Controller

   d) @Component

   Answer: c)

4. In Spring AOP, what is a "Join Point"?

   a) The main class of a project

   b) A place in the program where an aspect can be applied

   c) The configuration file for database connectivity

   d) A unit test function

   Answer: b)

5. Which of the following is **NOT** a Spring Boot starter dependency?

   a) spring-boot-starter-web

   b) spring-boot-starter-data-jpa

   c) spring-boot-starter-security

   d) spring-boot-starter-html

   Answer: d)

DescrFiptive Questions

1. Describe the purpose of **Spring JDBC** and explain how JdbcTemplate simplifies database operations.

2. Compare and contrast **Spring Framework** and **Spring Boot**. How does Spring Boot simplify application development?

3. Illustrate the **Spring Boot architecture** and explain the role of DispatcherServlet, Controller, and ViewResolver.

4. What are the main advantages of using **annotations** in Spring Boot for dependency management and configuration?

5. Define **Aspect-Oriented Programming (AOP)** in Spring Boot. Explain the types of advice supported and their use cases.

18.14: References and Suggested Readings

- Rod Johnson et al., *Spring Framework Reference Documentation*, Pivotal Software, 2024.

- Craig Walls, *Spring Boot in Action*, Manning Publications, 2023.

- Josh Long and Mark Heckler, *Cloud Native Java: Designing Resilient Systems with Spring Boot*, O'Reilly Media, 2022.

- Oracle, *Java EE and JDBC Documentation*, Oracle Corporation, 2024.

- Baeldung Tutorials, *Spring and Spring Boot Guides*, https://www.baeldung.com/

- Spring.io Documentation, *Spring Boot Official Reference Guide*, https://docs.spring.io/spring-boot/

- Kathy Sierra & Bert Bates, *Head First Java*, O'Reilly Media, 2023.

Glossary

Abstraction: A concept in OOP that hides int ernal implementation details and shows only functionality to the user.

Access Modifiers: Keywords used to set the access level for class es, methods, and variables (e.g., `public`, `private`, `protected`).

Animation: In JavaFX, animation involves changing property values over time to create motion or visual effects.

Annotation: Metadata in Java used to provide information to the compiler and frameworks like Spring (e.g., `@Component`, `@Autowired`).

Application Context: The central int erface in Spring for providing configuration and managing beans.

Bean: An object managed by the Spring IoC container.

Binding: A feature in JavaFX that links properties so that a change in one automatically updates the other.

Catch Block: Used in exception handling to catch and handle exceptions.

Class : A user-defined blueprint or prototype from which objects are created.

Component: A Spring stereotype annotation used to indicate that a class is a Spring-managed bean.

Constructor: A special method used to initialize objects.

CSS in JavaFX: Cascading Style Sheets used to style JavaFX UI elements.

Dependency Injection (DI): A design pattern in Spring where objects are passed their dependencies rather than creating them.

Directive Tag: JSP element that provides global information (e.g., `page`, `include`, `taglib`).

Encapsulation: Wrapping code and data int o a single unit, usually a class .

Exception: An event that disrupts normal program flow, handled using try-catch blocks.

Expression Language (EL): A JSP feature to simplify the access of Java objects and attributes without using Java code.

FXML: An XML-based language used to define the JavaFX UI structure.

finally Block: Executed after the try and catch blocks regardless of exception occurrence.

FX Effects: Visual effects in JavaFX like shadow, bloom, and reflection.

FX Shapes: 2D shapes such as rectangles, circles, and lines used in JavaFX applications.

HashMap: A collection class in Java that stores data in key-value pairs.

Implicit Objects: Predefined JSP objects like `request`, `response`, `session`, `application`, etc.

init() Method: Servlet lifecycle method called once during initialization.

Inversion of Control (IoC): A principle where control over object creation is transferred from the program to a container or framework like Spring.

JavaFX: A GUI toolkit for developing rich int ernet applications using Java.

Java Package: A namespace for organizing Java class es and int erfaces.

JSP (JavaServer Pages): A server-side technology that allows embedding Java code in HTML pages for dynamic content.

JSP Life Cycle: The process involving translation, compilation, initialization, execution, and destruction of a JSP page.

J2EE (Java 2 Enterprise Edition): A platform for developing multi-tier enterprise-level applications.

KeyFrame: In JavaFX, represents a specific point in time in a Timeline animation.

Method Overloading: Defining multiple methods in the same class with the same name but different parameters.

Method Overriding: Providing a new implementation of a method in a subclass that is already defined in its superclass .

Model-View-Controller (MVC): A design pattern that separates business logic, UI, and input handling.

Multithreading: A process of executing multiple threads simultaneously to perform multiple tasks.

Node: The base class for all JavaFX scene graph objects.

Object: An instance of a class that contains state and behavior.

Package: A grouping of related class es and int erfaces.

Pane: A layout container in JavaFX used to organize UI elements.

Polymorphism: An OOP concept that allows a method or object to behave in different ways.

Request Object: In Servlets and JSP, it represents the client's request and provides data like parameters and headers.

Response Object: Sends data back to the client from the server.

Runnable Int erface: Used to define a task that can be executed by a thread.

Scene: In JavaFX, a container for all content in a window (stage).

Scene Graph: A hierarchical tree structure in JavaFX where each element is a node.

Servlet: A Java class used to handle HTTP requests and responses in web applications.

Servlet Container: Software that manages servlets, like Apache Tomcat.

Servlet Life Cycle: Consists of `init()`, `service()`, and `destroy()` methods.

Spring Boot: An extension of the Spring framework that simplifies development with auto-configuration and embedded servers.

Spring Framework: A comprehensive framework for building Java applications with features like DI, AOP, and transaction management.

Stage: The top-level JavaFX container representing a window.

Static Keyword: Used for defining class -level variables and methods.

super Keyword: Used to call the constructor or methods of a parent class .

Scripting Elements (JSP): Include scriptlets (`<% %>`), expressions (`<%= %>`), and declarations (`<%! %>`).

Synchronization: A mechanism in Java to control access of multiple threads to shared resources.

Text Class : Used in JavaFX to display a String of text.

Thread: A lightweight subprocess for concurrent execution.

Thread Lifecycle: Includes New , Runnable, Running, Blocked, and Terminated states.

this Keyword: Refers to the current object in a method or constructor.

Thymeleaf: A Java template engine commonly used with Spring Boot for rendering HTML pages.

Timeline: In JavaFX, used to create animations over time.

Transformations (JavaFX): Operations like scaling, translating, and rotating nodes.

try Block: Used to enclose code that may throw exceptions.

wait(), notify(), notifyAll(): Methods for int er-thread communication in Java.

@SpringBootApplication: A convenience annotation for Spring Boot applications.

WebApplicationInitializer: Int erface for configuring Spring applications in a servlet container.

@Autowired: Spring annotation used to inject bean dependencies automatically.

@Controller: Annotation that marks a class as a web controller in Spring MVC.

Java Initializer: Used to bootstrap Spring Boot projects with selected dependencies.

MATS UNIVERSITY

MATS CENTRE FOR DISTANCE AND ONLINE EDUCATION

UNIVERSITY CAMPUS: Aarang Kharora Highway, Aarang, Raipur, CG, 493 441

RAIPUR CAMPUS: MATS Tower, Pandri, Raipur, CG, 492 002

T : 0771 4078994, 95, 96, 98 **Toll Free ODL MODE :** 81520 79999, 81520 29999

Website: www.matsodl.com