OOP in Java BCA 3rd

Topic: Introduction to Java

Introduction to Java

Java is a high-level, versatile, and widely used programming language that is known for its portability, efficiency, and ease of use in building a wide range of applications—from simple mobile apps to large-scale enterprise systems. Java is one of the most popular programming languages today and plays a significant role in modern computing.

1. Definition of Java:

Java is an object-oriented, class-based, and concurrent programming language. It was designed to have as few implementation dependencies as possible, making it suitable for developing cross-platform applications. Java enables developers to write code once and run it anywhere, thanks to its ability to execute on any system that supports the Java Virtual Machine (JVM). Java was initially developed by Sun Microsystems in 1995, and it has grown to be a primary language for web applications, mobile apps (especially Android), and large enterprise systems.

2. History of Java:

Java was created by James Gosling and Mike Sheridan at Sun Microsystems in 1991 under the original name *Oak*. The language was intended to be a simple, platform-independent language that could be used for embedded systems. It was renamed Java in 1995 and became an essential part of Sun Microsystems' platform strategy. Java's slogan, "Write Once, Run Anywhere" (WORA), refers to its portability due to its architecture-independent nature.

- **1991:** Java was born at Sun Microsystems.
- **1995:** Officially released as Java 1.0 and quickly became popular with web developers.
- **1997:** Java was standardized by the International Organization for Standardization (ISO) and the European Computer Manufacturers Association (ECMA).
- **2009:** Oracle acquired Sun Microsystems, and Java became part of Oracle's software stack.

3. The Internet and Java's Place in IT:

Java has played a crucial role in the rise of the Internet and has become an essential technology for building dynamic, interactive websites and web applications. With the advent of web-based applications in the late 1990s, Java became central to the development of server-side technologies, such as JSP (JavaServer Pages) and servlets, making it a vital tool for building large-scale web services.

Java's platform independence means that Java applications can run on any device that supports the JVM, whether that's a Windows machine, a Mac, or a server running Linux. Java's robustness, security, and scalability make it a key technology in today's IT infrastructure.

4. Applications and Applets:

Java is widely used in several areas of software development, including:

- **Web Applications:** Java's robust libraries and frameworks (e.g., Spring, Hibernate) make it an excellent choice for developing complex web applications.
- **Enterprise Applications:** Java Enterprise Edition (JEE) offers a comprehensive platform for building large-scale business applications.
- **Mobile Applications:** Java is the primary language used for Android app development, thanks to Android's reliance on the Java-based Android SDK (Software Development Kit).
- **Embedded Systems:** Java's portability makes it ideal for embedded systems, from smart devices to IoT applications.

Java Applets: Java Applets were small applications that could be embedded in web pages and run in a browser. However, applets have largely fallen out of favor due to security concerns and the development of more modern web technologies like HTML5, JavaScript, and CSS. Despite this, they were once widely used to create interactive content on the web.

5. Java Virtual Machine (JVM):

The JVM is a critical component of the Java platform. It is responsible for running Java programs by converting Java bytecode into machine code. The JVM acts as an intermediary between the Java program and the underlying operating system. The key advantage of the JVM is its portability: Java bytecode can be run on any platform with a JVM installed, enabling Java's "Write Once, Run Anywhere" promise.

6. Bytecode – Not an Executable Code:

When a Java program is compiled, it does not produce machine code directly. Instead, the Java compiler generates *bytecode*. Bytecode is an intermediate, platform-independent code that can be interpreted or compiled by the JVM at runtime. This means that Java code can be executed on any system that has a JVM, without modification.

- **Compilation Process:** The Java source code is first compiled into bytecode (.class files) using the Java compiler (javac).
- **Execution Process:** The bytecode is then executed by the JVM, which translates it into machine code for the specific system.

7. Procedure-Oriented vs. Object-Oriented Programming:

- **Procedure-Oriented Programming:** In procedure-oriented programming (POP), the focus is on functions or procedures that operate on data. The code is written as a series of step-by-step instructions. Languages such as C are often used for procedural programming.
- **Object-Oriented Programming (OOP):** In contrast, Java is an object-oriented programming language, meaning that it organizes code around *objects* rather than actions and data. An object is an instance of a class, which is a blueprint that defines the properties (data) and behaviors (methods) of an object. OOP promotes code reusability, scalability, and maintainability. Key principles of OOP include:

   - **Encapsulation:** Bundling data and methods into a single unit (class).
   - **Inheritance:** Allowing one class to inherit properties and behaviors from another class.
   - **Polymorphism:** Enabling different classes to be treated as instances of the same class through interfaces or abstract classes.
   - **Abstraction:** Hiding complex implementation details and exposing only necessary parts.

8. Compiling and Running a Simple Program:

A simple Java program requires the following steps to be compiled and run:

1. **Writing the Code:**
   - Use any text editor or Integrated Development Environment (IDE) such as Eclipse or IntelliJ IDEA to write Java code.
   - Example code:

   ```
   public class HelloWorld {
     public static void main(String[] args) {
       System.out.println("Hello, World!");
     }
   }
   ```
2. **Compiling the Program:**
   - Use the javac command to compile the Java code into bytecode.
   - Command: javac HelloWorld.java
   - This produces a file named HelloWorld.class.
3. **Running the Program:**
   - Use the java command to run the compiled bytecode on the JVM.
   - Command: java HelloWorld
   - This will print “Hello, World!” to the console.

9. Setting up Your Computer for Java Environment:

To start developing Java applications, you need to set up the Java Development Kit (JDK) on your computer. The JDK includes tools for compiling and running Java applications.

Steps to set up:

1. **Download and Install JDK:** Go to the official Oracle website and download the JDK version suitable for your operating system.

2. **Set Environment Variables:**
    - Set the JAVA_HOME variable to the directory where the JDK is installed.
    - Add the JDK bin directory to your system's PATH variable to run Java commands from the command line.
3. **Install IDE (Optional but Recommended):** Download and install an IDE like IntelliJ IDEA, Eclipse, or NetBeans, which provides a more user-friendly environment for coding and debugging Java applications.

10. Writing a Program:

Once the environment is set up, you can start writing Java programs. A typical Java program consists of a class (or classes) with methods that define the behavior of the program. Here's an example of a basic program:

```
public class Example {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

11. Compiling, Interpreting, and Running the Program:

- **Compiling:** Using the javac command, the Java compiler translates the .java file into bytecode (.class).
- **Interpreting/Running:** The JVM reads the bytecode and executes it. The interpreter or JIT (Just-In-Time) compiler within the JVM converts bytecode into machine code suitable for the target system.

12. Handling Common Errors:

Common errors in Java programming are typically divided into three categories:

- **Syntax Errors:** These occur when the Java code does not follow proper syntax, like missing semicolons or mismatched brackets.
    - Example: System.out.println("Hello World") (missing semicolon).
- **Runtime Errors:** These occur during program execution, often due to improper input or logic.
    - Example: Division by zero.
- **Logical Errors:** These errors occur when the program runs but doesn't produce the expected result, usually due to incorrect algorithm design or logic flaws.

In Java, proper error handling is essential. The try-catch block is used to catch and handle exceptions.

```
try {
   int result = 10 / 0; // Causes ArithmeticException
} catch (ArithmeticException e) {
   System.out.println("Error: Division by zero!");
}
```

Conclusion

Java is a versatile and widely-used language with a rich history and a solid foundation in object-oriented principles. With its platform independence, robust libraries, and focus on security, Java continues to be a critical tool for software development, especially in enterprise, web, and mobile applications. Understanding the basics of Java, from setting up the environment to handling common errors, is the first step toward mastering this powerful programming language.

Topic: Tokens, Expressions and Control Structures

Note on Tokens, Expressions, and Control Structures (5 hours)

1. Primitive Data Types

Primitive data types are the basic types that represent simple values. These are typically built into the programming language and are not derived from other data types. Common primitive data types include:

- **Integers**: Represent whole numbers (e.g., int in C or Java).
- **Floating-Point types**: Represent real numbers (e.g., float or double in C/C++ or Java).
- **Characters**: Represent single characters (e.g., char in C/C++ or Java).
- **Booleans**: Represent true or false values (e.g., boolean in Java or bool in C++).

2. User-Defined Data Types

User-defined data types allow programmers to create new types based on existing primitive data types, which can be used to structure data. These include:

- **Structures** (C/C++), **Classes** (in Object-Oriented Programming languages), **Enumerations** (enums), and **Unions**.

3. Declarations, Constants, Identifiers, and Literals

- **Declarations**: Define variables and their data types in code, e.g., int x;.
- **Constants**: A value that cannot be modified after its initial assignment, often defined using keywords like const (C/C++) or final (Java).

- **Identifiers**: Names used to identify variables, functions, arrays, classes, etc. Identifiers must follow naming conventions (e.g., no spaces, cannot begin with a digit).
- **Literals**: Fixed values used in code (e.g., 5, "hello", true).

4. Type Conversion and Casting

- **Type Conversion**: The automatic or manual conversion between different data types, such as converting an int to a float.
- **Casting**: Explicit conversion of one data type to another using type casting (e.g., (int) 3.14 to convert 3.14 to an integer).

5. Variables: Definition, Assignment, and Initialization

- **Variable Definition**: Declaring a variable with a specific data type (e.g., int x;).
- **Assignment**: Assigning a value to a variable (e.g., x = 10;).
- **Default Initialization**: In some languages (like Java), variables are initialized with default values (e.g., int defaults to 0).

6. Command-Line Arguments

- Command-line arguments are passed to the program when it is executed from the terminal. In languages like C/C++, they are accessible via the main function parameters (argc and argv).

7. Arrays of Primitive Data Types

- Arrays are collections of variables of the same data type, such as an array of integers int arr[5]. Arrays are essential for storing and manipulating multiple values.

8. Comment Syntax

- Comments are used for documentation and explaining the code.
    - Single-line comments: // This is a comment
    - Multi-line comments: /* This is a comment */

9. Garbage Collection

- Garbage collection refers to automatic memory management in languages like Java or Python, where the system automatically frees memory used by objects that are no longer needed, helping to prevent memory leaks.

10. Expressions and Operators

- **Expressions**: An expression is a combination of variables, constants, operators, and function calls that evaluates to a value (e.g., x + y).
- **Operators**:
    - **Arithmetic Operators**: Used for mathematical operations (+, -, *, /, %).

- **Bitwise Operators**: Used to perform operations on binary representations (&, |, ^, <<, >>).
- **Relational Operators**: Used to compare values (==, !=, <, >, <=, >=).
- **Logical Operators**: Used to perform logical operations (&&, ||, !).
- **Assignment Operators**: Used to assign values to variables (=, +=, -=, *=, /=, etc.).
- **Conditional Operator**: A shorthand for if-else statements, written as condition ? expr1 : expr2.
- **Shift Operators**: Used to shift bits to the left or right (<<, >>).
- **Auto-increment/Decrement Operators**: Increment or decrement a variable by 1 (++, --).

11. Control Statements

Control statements define the flow of execution of a program. There are three main categories:

- **Branching Statements**: Direct the flow based on conditions.
  - if: Executes code based on a condition (if (x > 0) { ... }).
  - switch: Provides a multi-way branching structure based on matching values (switch (x) { case 1: ...; break; }).
- **Looping Statements**: Repeat a block of code as long as a condition is true.
  - while: Executes a block of code as long as the condition is true (while (x < 10) { ... }).
  - do-while: Similar to while, but guarantees that the code block runs at least once (do { ... } while (x < 10);).
  - for: Typically used when the number of iterations is known (for (int i = 0; i < 10; i++) { ... }).
- **Jumping Statements**: Alter the flow of control within loops or functions.
  - break: Exits from a loop or switch statement prematurely.
  - continue: Skips the current iteration of a loop and proceeds with the next iteration.
  - return: Exits from a function and optionally returns a value.

Conclusion

In programming, understanding tokens, expressions, and control structures is essential for building functional and efficient programs. Mastering primitive data types, operators, control flow, and variable management equips programmers to solve problems logically and precisely.

Topic: Object Oriented Programming Concepts

Sure! Below is a more detailed explanation of Object-Oriented Programming (OOP) concepts with examples in Java.

Object-Oriented Programming Concepts (9 hours)

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes to structure software. It revolves around the idea of grouping data and the methods that operate on that data into one unit called a **class**.

The main OOP principles are **Abstraction**, **Encapsulation**, **Polymorphism**, and **Inheritance**. Let's go over each of these concepts with practical Java examples.

---

1. Fundamentals of Classes in Java

A **class** in Java is a blueprint for objects. It defines properties (variables) and behaviors (methods) that the objects created from the class will have.

A. Simple Class

In Java, a class can contain fields (data members) and methods (functions) to operate on the data.

```
// Simple class in Java

class Car {

   String brand;  // Data member (property)

   int year;      // Data member (property)



   // Method inside the class

   void displayInfo() {

      System.out.println("Car brand: " + brand + ", Year: " + year);

   }

}
```

B. Creating Class Instances

After defining the class, you can create instances (objects) of the class using the new keyword.

```
public class Main {

   public static void main(String[] args) {

      // Creating an object of the Car class
```

```
        Car myCar = new Car();
        myCar.brand = "Toyota";
        myCar.year = 2020;

        // Calling a method using the object
        myCar.displayInfo();  // Output: Car brand: Toyota, Year: 2020
    }
}
```

Here, myCar is an object of the Car class. Each object can have its own properties like brand and year.

---

2. Adding Methods to a Class

Methods define the behavior of a class. In Java, methods can be used to perform actions or manipulate the properties of the object.

```
class Car {
    String brand;
    int year;

    // Method to set the values
    void setDetails(String brand, int year) {
        this.brand = brand;  // Using 'this' to refer to the instance variable
        this.year = year;
    }

    // Method to display the details
    void displayInfo() {
        System.out.println("Car brand: " + brand + ", Year: " + year);
    }
}
```

In this example, setDetails() sets the brand and year fields of the Car class, and displayInfo() prints them to the console.

---

3. Abstraction in Java

Abstraction is the concept of hiding implementation details and showing only essential features of an object. In Java, abstraction is achieved using **abstract classes** and **interfaces**.

Abstract Class Example:

```
abstract class Shape {
    abstract void draw();  // Abstract method without implementation
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a circle");
    }
}

class Square extends Shape {
    void draw() {
        System.out.println("Drawing a square");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle();
        circle.draw();  // Output: Drawing a circle

        Shape square = new Square();
        square.draw();  // Output: Drawing a square
    }
}
```

In this example:

- Shape is an abstract class with an abstract method draw().
- Circle and Square are concrete subclasses that implement the draw() method.

---

4. Encapsulation in Java

Encapsulation is the practice of hiding the internal state of an object and restricting access to it. In Java, this is achieved by using **private** fields and providing **public getter and setter methods**.

```
class Car {
    private String brand;  // Private field
    private int year;      // Private field

    // Public setter method
    public void setBrand(String brand) {
        this.brand = brand;
    }

    // Public getter method
    public String getBrand() {
        return brand;
    }

    public void setYear(int year) {
        this.year = year;
    }

    public int getYear() {
        return year;
    }
}


public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.setBrand("Honda");
        myCar.setYear(2021);


        System.out.println("Car brand: " + myCar.getBrand());
        System.out.println("Car year: " + myCar.getYear());
    }
}
```

In this example:

- The brand and year fields are private, so they cannot be accessed directly outside the class.
- The setter and getter methods provide controlled access to these fields.

---

5. Using the this Keyword

The this keyword refers to the current instance of the class. It helps distinguish between class fields and method parameters when they have the same name.

```
class Car {
    String brand;
    int year;


    // Constructor using 'this' to refer to instance variables
    Car(String brand, int year) {
        this.brand = brand;  // 'this' refers to the current object
        this.year = year;
    }


    void displayInfo() {
        System.out.println("Car brand: " + this.brand + ", Year: " + this.year);
    }
}


public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Ford", 2020);
        myCar.displayInfo();  // Output: Car brand: Ford, Year: 2020
    }
}
```

Here, this.brand and this.year refer to the instance variables of the Car class, and brand and year refer to the constructor parameters.

---

6. Constructors and Default Constructors

A **constructor** is a special method used to initialize an object when it is created. Java provides a **default constructor** if no constructors are defined.

Default Constructor Example:

```
class Car {
    String brand;
    int year;


    // Default constructor
    Car() {
        this.brand = "Unknown";
        this.year = 0;
    }
    void displayInfo() {
        System.out.println("Car brand: " + brand + ", Year: " + year);
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();  // Default constructor is called
        myCar.displayInfo();  // Output: Car brand: Unknown, Year: 0
    }
}
```

In this example, the default constructor initializes brand to "Unknown" and year to 0.

---

7. Polymorphism in Java

Polymorphism allows objects of different types to be treated as objects of a common superclass. There are two types of polymorphism:

- **Method Overloading** (compile-time polymorphism)
- **Method Overriding** (runtime polymorphism)

A. Method Overloading (Compile-time Polymorphism)

Method overloading allows multiple methods with the same name but different parameters.

```
class Car {
    void displayInfo(String brand) {
        System.out.println("Car brand: " + brand);
    }

    void displayInfo(String brand, int year) {
        System.out.println("Car brand: " + brand + ", Year: " + year);
    }
}


public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.displayInfo("Toyota");  // Output: Car brand: Toyota
        myCar.displayInfo("Honda", 2020);  // Output: Car brand: Honda, Year: 2020
    }
}
```

In this example, displayInfo() is overloaded with two versions: one that takes a single parameter and one that takes two parameters.

B. Method Overriding (Runtime Polymorphism)

Method overriding occurs when a subclass provides its own implementation of a method that is already defined in the superclass.

```
class Shape {
    void draw() {
        System.out.println("Drawing a shape");
    }
}


class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a circle");
    }
}


public class Main {
    public static void main(String[] args) {
        Shape shape = new Circle();
        shape.draw();  // Output: Drawing a circle
    }
}
```

Here, the draw() method is overridden in the Circle class. Even though the reference type is Shape, the method of the Circle class is called, demonstrating polymorphism.

---

8. Recursion in Java

Recursion occurs when a method calls itself. It is typically used for problems that can be broken down into smaller subproblems, such as computing factorials or Fibonacci numbers.

```
class Math {
    // Recursive method to calculate factorial
    int factorial(int n) {
        if (n == 0) return 1;
        else return n * factorial(n - 1);
    }
}

public class Main {
    public static void main(String[] args) {
        Math math = new Math();
        System.out.println("Factorial of 5: " + math.factorial(5));  // Output: 120
    }
}
```

In this example, the factorial() method calls itself to compute the factorial of a number.

---

9. Nested and Inner Classes

Java supports **nested classes** (classes defined inside other classes). **Inner classes** are non-static nested classes that can access instance members of the outer class.

Nested Class Example:

```
class Outer {
    class Inner {
        void display() {
            System.out.println("Inner class method");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.display();  // Output: Inner class method
    }
}
```

In this example, Inner is a non-static nested (inner) class inside Outer. An instance of the Inner class can be created using an instance of the Outer class.

Conclusion

In Object-Oriented Programming (OOP), Java provides a rich set of features that help in designing modular, reusable, and maintainable software. The core OOP principles such as **Abstraction**, **Encapsulation**, **Polymorphism**, and **Inheritance** are implemented in Java through classes, objects, methods, constructors, and other powerful features. Mastering these concepts enables developers to build complex software with clear, organized code.

Topic: Inheritance & Packaging

Inheritance & Packaging in Java (3 hours)

In Object-Oriented Programming (OOP), **Inheritance** allows a class to inherit properties and behaviors (fields and methods) from another class. It helps promote code reusability, extensibility, and organization. In Java, inheritance is implemented using the extends keyword. This concept is closely related to the ideas of **subclasses** and **superclasses**, and Java provides several mechanisms to manage access control, method overriding, dynamic dispatch, and more.

In addition to inheritance, Java supports **packages**, **interfaces**, and access control mechanisms, which are essential to organize and modularize code. Let's explore these concepts in detail.

---

1. Inheritance in Java

A. Using the extends Keyword

Inheritance in Java allows one class (called the **subclass** or **child class**) to inherit the properties and methods of another class (called the **superclass** or **parent class**). The subclass extends the superclass using the extends keyword.

Example: Using the extends Keyword

```
// Superclass (Parent class)
class Animal {
    String name;

    // Constructor
    Animal(String name) {
        this.name = name;
    }

    // Method of superclass
    void sound() {
        System.out.println("The animal makes a sound");
    }
}

// Subclass (Child class)
class Dog extends Animal {
    // Constructor of subclass
    Dog(String name) {
        super(name);  // Calls the superclass constructor
    }
    // Method overriding
    @Override
    void sound() {
        System.out.println("The dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        dog.sound();  // Output: The dog barks
    }
}
```

In this example:

- Dog extends Animal, which means Dog inherits all fields and methods of Animal.
- The sound() method is **overridden** in the Dog class.

B. Subclasses and Superclasses

- **Superclass**: A class that is inherited by other classes (parent class).
- **Subclass**: A class that inherits from another class (child class).

C. Using the super Keyword

The super keyword in Java is used to refer to the superclass. It allows the subclass to call the superclass's methods and constructors.

Example: Using super Keyword

```
// Superclass (Parent class)
class Animal {
    String name;

    // Constructor of superclass
    Animal(String name) {
        this.name = name;
    }

    // Method of superclass
    void sound() {
        System.out.println("The animal makes a sound");
    }
}

// Subclass (Child class)
class Dog extends Animal {
    Dog(String name) {
        super(name);  // Calls the constructor of the superclass
    }

    @Override
    void sound() {
        super.sound();  // Calls the superclass method
        System.out.println("The dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        dog.sound();  // Output: The animal makes a sound
                      //         The dog barks
    }
```

```
}
```

In this example:

- The super() constructor call is used to initialize the name in the Animal class.
- The super.sound() calls the sound() method of the superclass.

D. Overriding Methods

Method overriding occurs when a subclass provides its specific implementation for a method that is already defined in the superclass. In Java, the @Override annotation is used to indicate method overriding.

Example: Method Overriding

```
class Animal {
    void sound() {
        System.out.println("The animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("The dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();
        animal.sound();  // Output: The dog barks (Dynamic method dispatch)
    }
}
```

- Here, the Dog class overrides the sound() method.

- Dynamic method dispatch allows the appropriate method to be called based on the actual object type (Dog) at runtime, even though the reference type is Animal.

---

2. The Object Class

In Java, every class implicitly inherits from the Object class. The Object class is the root of the class hierarchy in Java, and it provides fundamental methods such as equals(), hashCode(), toString(), and clone().

Example: Using the Object Class Methods

```
class Car {
    String brand;


    Car(String brand) {
        this.brand = brand;
    }


    @Override
    public String toString() {
        return "Car brand: " + brand;
    }
}


public class Main {
    public static void main(String[] args) {
        Car car = new Car("Toyota");
        System.out.println(car.toString());  // Output: Car brand: Toyota
    }
}
```

In this example:

- The Car class overrides the toString() method from the Object class to return a custom string representation of the object.

---

3. Abstract and Final Classes

A. Abstract Classes

An **abstract class** cannot be instantiated directly. It may contain abstract methods (methods without a body) that must be implemented by subclasses.

Example: Abstract Class

```
abstract class Animal {
  String name;

  Animal(String name) {
    this.name = name;
  }

  // Abstract method (no body)
  abstract void sound();
}

class Dog extends Animal {
  Dog(String name) {
    super(name);
  }

  // Implementing the abstract method
  @Override
  void sound() {
    System.out.println("The dog barks");
  }
}
```

```
public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog("Buddy");
        dog.sound();  // Output: The dog barks
    }
}
```

- Animal is an abstract class with an abstract method sound().
- Dog provides an implementation for sound().

B. Final Classes

A **final class** cannot be extended. It cannot have any subclasses.

Example: Final Class

```
final class Car {
    String brand;

    Car(String brand) {
        this.brand = brand;
    }

    void displayInfo() {
        System.out.println("Car brand: " + brand);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car("Toyota");
        car.displayInfo();  // Output: Car brand: Toyota
    }
}
```

- Car is a final class, so it cannot be extended by any other class.

---

4. Package in Java

A **package** in Java is a way to group related classes and interfaces together. It helps in organizing the code, avoiding naming conflicts, and controlling access.

A. Access Control

In Java, access control modifiers define the visibility and accessibility of classes, methods, and variables:

- **public**: The member is accessible from any other class.
- **private**: The member is only accessible within the same class.
- **protected**: The member is accessible within the same package or subclasses.
- **Default (no modifier)**: The member is accessible only within the same package.

Example: Access Control in Package

```
package vehicle;


public class Car {
  private String brand;
  protected int year;


  public Car(String brand, int year) {
    this.brand = brand;
    this.year = year;
  }


  public void displayInfo() {
    System.out.println("Brand: " + brand + ", Year: " + year);
  }
}
```

- Car is part of the vehicle package. The brand variable is private, so it is not accessible outside the class, but year is protected and can be accessed by subclasses.

5. Interfaces in Java

An **interface** in Java is a contract that specifies a set of methods that a class must implement. It is similar to an abstract class but can only contain abstract methods (no implementation) and constants.

A. Defining an Interface

```
interface Animal {
   void sound();  // Abstract method
}

class Dog implements Animal {
   @Override
   public void sound() {
      System.out.println("The dog barks");
   }
}

public class Main {
   public static void main(String[] args) {
      Animal animal = new Dog();
      animal.sound();  // Output: The dog barks
   }
}
```

- Animal is an interface with an abstract method sound().
- Dog implements the Animal interface and provides an implementation for the sound() method.

B. Applying Multiple Interfaces

Java allows a class to implement multiple interfaces, which helps in achieving multiple inheritance of behavior.

```
interface Flyable {
   void fly();
}

interface Swimmable {
  void swim();
}

class Duck implements Flyable, Swimmable {
  @Override
  public void fly() {
    System.out.println("The duck flies");
  }

  @Override
  public void swim() {
    System.out.println("The duck swims");
  }
}

public class Main {
  public static void main(String[] args) {
    Duck duck = new Duck();
    duck.fly();   // Output: The duck flies
    duck.swim();  // Output: The duck swims
  }
}
```

- Duck implements both Flyable and Swimmable interfaces, providing implementations for both fly() and swim().

Conclusion

In Java, **inheritance** allows classes to inherit properties and behaviors from other classes, promoting code reuse. The extends keyword is used to create subclasses that inherit from superclasses, and the super keyword helps call superclass methods and constructors. Java also supports **abstract** and **final** classes to control inheritance behavior.

Packages provide a way to organize related classes and control access to them using access modifiers. **Interfaces** define contracts that classes must

Topic: Handling Error / Exceptions

Handling Errors/Exceptions in Java (2 hours)

In Java, **exceptions** are events that disrupt the normal flow of execution in a program. They are used to handle errors or unexpected conditions. Java provides a robust exception-handling mechanism that helps programmers deal with runtime errors, making the program more reliable and fault-tolerant.

The **exception handling** mechanism in Java is based on the try, catch, throw, throws, and finally keywords, allowing you to catch and handle exceptions, throw exceptions, and ensure that certain cleanup actions are performed regardless of exceptions.

---

1. Basic Exceptions in Java

Java has a built-in class hierarchy for exceptions, which extends the Throwable class. There are two main types of exceptions:

- **Checked Exceptions**: These are exceptions that are checked at compile-time, such as IOException, SQLException, etc.
- **Unchecked Exceptions**: These are exceptions that occur during runtime and are subclasses of RuntimeException, such as NullPointerException, ArithmeticException, etc.

Example of Basic Exception Handling:

```
public class Main {

   public static void main(String[] args) {

      try {

         int result = 10 / 0;  // This will throw ArithmeticException

      } catch (ArithmeticException e) {

         System.out.println("Error: Division by zero!");

      }

   }
}
```

- In this example, we have a division by zero, which will throw an ArithmeticException. The catch block catches the exception and prints a meaningful message.

---

2. Proper Use of Exceptions

Exceptions should be used for **exceptional conditions** (unpredictable or unusual situations), not for regular flow control. In Java, exceptions can be caught and handled at various levels using try, catch, throw, throws, and finally.

A. Catching Exceptions with try and catch

The try block contains the code that may cause an exception, and the catch block handles the exception.

Example: Catching Multiple Exceptions

```
public class Main {
  public static void main(String[] args) {
    try {
      String[] arr = new String[3];
      arr[4] = "Hello";  // ArrayIndexOutOfBoundsException

      int result = 10 / 0;  // ArithmeticException
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Error: Array index out of bounds!");
    } catch (ArithmeticException e) {
      System.out.println("Error: Division by zero!");
    }
  }
}
```

- The try block contains code that could throw either an ArrayIndexOutOfBoundsException or ArithmeticException.
- Both exceptions are caught in separate catch blocks, and appropriate error messages are displayed.

B. Using finally for Cleanup

The finally block contains code that is always executed, regardless of whether an exception occurs or not. It is typically used for **resource cleanup** (e.g., closing files, database connections).

Example: Using finally for Cleanup

```
public class Main {
   public static void main(String[] args) {
      try {
         System.out.println("Attempting to divide...");
         int result = 10 / 0;
      } catch (ArithmeticException e) {
         System.out.println("Error: Division by zero!");
      } finally {
         System.out.println("Cleanup: This will always execute.");
      }
   }
}
```

- The finally block is always executed after the try and catch blocks, whether or not an exception occurs. It's commonly used for **resource cleanup** (e.g., closing files or database connections).

---

3. Throwing and Re-throwing Exceptions

In Java, exceptions can be explicitly thrown using the throw keyword. This allows you to create and throw custom exceptions or re-throw exceptions to propagate them further up the call stack.

A. Throwing Exceptions Using throw

The throw keyword is used to explicitly throw an exception from a method or a block of code.

Example: Throwing an Exception

```
public class Main {
   public static void checkAge(int age) {
      if (age < 18) {
         throw new IllegalArgumentException("Age must be 18 or older");
      }
      System.out.println("Age is valid");
```

```
    }


    public static void main(String[] args) {
        try {
            checkAge(15);  // This will throw an IllegalArgumentException
        } catch (IllegalArgumentException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

- In the checkAge() method, if the age is less than 18, an IllegalArgumentException is thrown.
- The main() method catches the exception and prints an error message.

B. Re-throwing Exceptions Using throws

The throws keyword is used in a method signature to indicate that a method may throw one or more exceptions. When a method calls another method that throws an exception, it can propagate the exception using throws.

Example: Re-throwing an Exception

```
public class Main {
    public static void riskyMethod() throws Exception {
        throw new Exception("Something went wrong");
    }


    public static void main(String[] args) {
        try {
            riskyMethod();
        } catch (Exception e) {
            System.out.println("Caught exception: " + e.getMessage());
        }
    }
}
```

- The riskyMethod() method throws an Exception which is propagated to the main() method using throws.
- The main() method catches the exception and handles it.

---

4. User-Defined Exceptions

In Java, you can create custom exceptions by extending the Exception class. These are known as **user-defined exceptions**. They can be used to represent specific error conditions in your application.

Example: User-Defined Exception

```
// Custom exception class
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class Main {
    public static void validateAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or older");
        }
        System.out.println("Valid age");
    }

    public static void main(String[] args) {
        try {
            validateAge(15);  // This will throw InvalidAgeException
        } catch (InvalidAgeException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

- InvalidAgeException is a custom exception class that extends the Exception class.
- The validateAge() method throws this custom exception if the age is less than 18.
- The main() method catches and handles the custom exception.

Conclusion

Java's **exception handling** mechanism provides a powerful and structured way to deal with errors and exceptional conditions in your programs. By using the try, catch, throw, throws, and finally keywords, you can:

- **Catch exceptions** and handle them gracefully.
- **Throw exceptions** when necessary to indicate errors or unusual conditions.
- **Clean up resources** in a finally block, ensuring certain actions (like closing files) always occur.
- **Create user-defined exceptions** to represent specific error conditions in your application.

Exception handling improves the robustness of your programs by allowing you to separate error handling from the main program logic and ensuring that errors are dealt with in a controlled manner.

Topic: Handling Strings

Handling Strings in Java (2 hours)

Strings in Java are objects that represent a sequence of characters. The **String** class is part of the java.lang package and provides several methods for creating, modifying, and manipulating string values. Java strings are immutable, meaning that once a string is created, it cannot be changed. However, there are various techniques available to work with strings efficiently, such as string concatenation, modification, and comparison.

In addition to the **String** class, Java also provides the **StringBuffer** class for handling mutable sequences of characters, which is more efficient for operations like appending or modifying strings frequently.

Let's dive into the different operations you can perform on strings in Java.

---

1. Creation of a String

Strings in Java can be created in two ways:

- Using string literals (which are stored in the string pool).

- Using the new keyword to create a new String object.

Example: Creating Strings

```
public class Main {
    public static void main(String[] args) {
        // Using string literal
        String str1 = "Hello, World!";

        // Using new keyword
        String str2 = new String("Hello, Java!");

        System.out.println(str1);
        System.out.println(str2);
    }
}
```

- str1 is created using a string literal.
- str2 is created using the new keyword.

---

2. Concatenation of Strings

You can concatenate (join) strings using the + operator or using the concat() method. The + operator is the most commonly used way, while concat() is a method of the String class that concatenates the specified string to the current string.

Example: String Concatenation

```
public class Main {
    public static void main(String[] args) {
        String str1 = "Hello, ";
        String str2 = "World!";

        // Using + operator
        String result1 = str1 + str2;
        System.out.println(result1);  // Output: Hello, World!

        // Using concat() method
        String result2 = str1.concat(str2);
        System.out.println(result2);  // Output: Hello, World!
    }
}
```

- Both methods produce the same result, but the + operator is more commonly used for simple concatenation.

---

3. Conversion of Strings

You can convert a string to different formats, such as converting a string to an array of characters, converting to uppercase or lowercase, or converting other data types to strings.

Example: String Conversion

```
public class Main {
    public static void main(String[] args) {
        // String to char array
        String str = "Hello";
        char[] charArray = str.toCharArray();
        System.out.println(charArray);  // Output: Hello

        // Convert to uppercase and lowercase
        String upper = str.toUpperCase();
        String lower = str.toLowerCase();

        System.out.println(upper);  // Output: HELLO
        System.out.println(lower);  // Output: hello
    }
}
```

- The toCharArray() method converts the string into an array of characters.
- The toUpperCase() and toLowerCase() methods return the string in uppercase or lowercase, respectively.

---

4. Changing Case of a String

You can easily change the case of a string using the toUpperCase() and toLowerCase() methods. These methods return a new string with the desired case.

Example: Changing Case

```
public class Main {
    public static void main(String[] args) {
        String str = "Java Programming";

        String upperStr = str.toUpperCase();
        String lowerStr = str.toLowerCase();

        System.out.println("Uppercase: " + upperStr);  // Output: JAVA PROGRAMMING
        System.out.println("Lowercase: " + lowerStr);  // Output: java programming
    }
}
```

- toUpperCase() converts the string to all uppercase letters.
- toLowerCase() converts the string to all lowercase letters.

---

5. Character Extraction from a String

You can extract characters from a string using methods like charAt() to get the character at a specific index, and substring() to get a part of the string.

Example: Character Extraction

```
public class Main {
    public static void main(String[] args) {
        String str = "Java Programming";
        // Extracting a single character
        char c = str.charAt(0);  // 'J'
        System.out.println("First character: " + c);  // Output: First character: J

        // Extracting a substring
        String subStr = str.substring(5, 16);  // Extracts "Programming"
        System.out.println("Substring: " + subStr);  // Output: Substring: Programming
    }
}
```

- charAt(index) returns the character at the specified index.
- substring(startIndex, endIndex) extracts a substring from the string starting from startIndex to endIndex-1.

---

6. String Comparison

Java provides several methods to compare strings, such as equals(), equalsIgnoreCase(), and compareTo().

Example: String Comparison

```
public class Main {
    public static void main(String[] args) {
        String str1 = "Java";
        String str2 = "java";
        String str3 = "Java";

        // Using equals() for case-sensitive comparison
        System.out.println(str1.equals(str2));  // Output: false
        System.out.println(str1.equals(str3));  // Output: true

        // Using equalsIgnoreCase() for case-insensitive comparison
        System.out.println(str1.equalsIgnoreCase(str2));  // Output: true
        // Using compareTo() for lexicographical comparison
        System.out.println(str1.compareTo(str2));  // Output: positive number (since 'J' > 'j')
        System.out.println(str1.compareTo(str3));  // Output: 0 (since both are equal)
    }
}
```

- equals() checks if two strings are equal, considering case.
- equalsIgnoreCase() compares two strings ignoring case.
- compareTo() compares two strings lexicographically and returns an integer based on the comparison.

---

7. Searching Strings

Java provides the indexOf() and contains() methods to search for a substring within a string.

Example: Searching in a String

```
public class Main {
    public static void main(String[] args) {
        String str = "Java Programming";

        // Checking if a substring exists
        System.out.println(str.contains("Java"));  // Output: true
        System.out.println(str.contains("Python"));  // Output: false

        // Finding the index of a substring
        System.out.println(str.indexOf("Pro"));  // Output: 5
        System.out.println(str.indexOf("xyz"));  // Output: -1 (not found)
    }
}
```

- contains() checks if a substring is present in the string and returns true or false.
- indexOf() returns the index of the first occurrence of a substring or -1 if not found.

---

8. Modifying Strings

Although strings in Java are immutable, you can create modified versions of strings using methods like replace(), replaceAll(), and replaceFirst().

Example: Modifying a String

```
public class Main {
   public static void main(String[] args) {
      String str = "Java Programming";

      // Replace characters
      String replacedStr = str.replace('a', 'o');  // Replaces all 'a' with 'o'
      System.out.println(replacedStr);  // Output: Jovo Progromming

      // Replace using regular expression
      String replacedRegexStr = str.replaceAll("a", "o");  // Same as above, replacing all 'a' with 'o'
      System.out.println(replacedRegexStr);  // Output: Jovo Progromming
   }
}
```

- replace() replaces all occurrences of a character with another character.
- replaceAll() replaces all occurrences matching a regular expression with a given replacement string.

---

9. StringBuffer

Unlike the immutable String, the **StringBuffer** class represents a mutable sequence of characters. It is used for efficient string manipulation when frequent modifications (like appending, inserting, or deleting characters) are required.

Example: Using StringBuffer

```
public class Main {
   public static void main(String[] args) {
      StringBuffer sb = new StringBuffer("Java");
```

```
        // Append string
        sb.append(" Programming");
        System.out.println(sb);  // Output: Java Programming

        // Insert string at a specific index
        sb.insert(4, " Language");
        System.out.println(sb);  // Output: Java Language Programming

        // Reverse the string
        sb.reverse();
        System.out.println(sb);  // Output: gnimmargorP egaugnaL avaJ
    }
}
```

- append() adds text to the end of the StringBuffer.
- insert() inserts text at a specified index.
- reverse() reverses the string.

Conclusion

Java provides a comprehensive set of tools to handle strings effectively:

- **String creation** is straightforward with string literals or the new keyword.
- **Concatenation**, **conversion**, and **modification** can be easily done using methods like concat(), toUpperCase(), substring(), and replace().
- String **comparison** can be performed using equals(), equalsIgnoreCase(), and compareTo().
- **StringBuffer** is ideal for frequent modifications to strings, as it provides more efficient memory management than regular strings.

Mastering string handling is essential for effective programming in Java, as strings are a central part of most applications.

Topic: Threads

Threads in Java (7 hours)

A **thread** in Java is a lightweight process that runs within a program. Java supports multithreading, which allows multiple threads to execute concurrently, improving the performance of CPU-bound tasks and enabling responsive applications. Threads are crucial for performing tasks like I/O operations, UI updates, and parallel computing.

Java provides several mechanisms for creating, managing, and controlling threads. These include extending the Thread class, implementing the Runnable interface, controlling thread priorities, synchronization, inter-thread communication, and avoiding issues like deadlock.

1. Creating and Instantiating Threads

There are two main ways to create threads in Java:

- **Extending the Thread class**
- **Implementing the Runnable interface**

A. Creating Threads by Extending the Thread Class

The simplest way to create a thread is by extending the Thread class. By overriding the run() method, you can define the task that the thread will perform when started.

Example: Extending Thread Class

```
class MyThread extends Thread {
   public void run() {
      System.out.println("Thread is running: " + Thread.currentThread().getName());
   }
}


public class Main {
   public static void main(String[] args) {
      MyThread t1 = new MyThread();  // Create thread object
      t1.start();  // Start the thread


      MyThread t2 = new MyThread();
      t2.start();
   }
}
```

- **MyThread class** extends Thread and overrides the run() method.
- **start()** is called on the thread object to begin execution, which internally invokes the run() method in a new thread.

B. Creating Threads by Implementing the Runnable Interface

Another way to create a thread is by implementing the Runnable interface. This approach is preferable if your class already extends another class (since Java supports single inheritance). The Runnable interface requires you to implement the run() method.

Example: Implementing Runnable Interface

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running using Runnable: " + Thread.currentThread().getName());
    }
}


public class Main {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();  // Create Runnable object
        Thread thread = new Thread(myRunnable);  // Pass Runnable to Thread
        thread.start();  // Start the thread
    }
}
```

- The run() method contains the code that will be executed by the thread.
- A Runnable instance is passed to a Thread constructor, and the start() method is invoked to start the thread.

---

2. Thread Execution and Lifecycle

A thread goes through several stages during its lifecycle:

1. **New**: A thread is created but not yet started.

2. **Runnable**: The thread is ready for execution, but it might be waiting for CPU resources.
3. **Blocked**: The thread is waiting to acquire a resource (e.g., waiting for I/O).
4. **Terminated**: The thread has finished execution.

Threads are managed by the Java Virtual Machine (JVM) and the operating system, and they may execute concurrently or sequentially, depending on system resources.

Example: Thread Lifecycle

```
class MyThread extends Thread {
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " is in RUNNING state.");
            Thread.sleep(1000);  // Simulate some work by sleeping for 1 second
            System.out.println(Thread.currentThread().getName() + " is in TERMINATED state.");
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

- The thread enters the RUNNING state, performs some work (sleep), and then moves to the TERMINATED state when it finishes.

---

3. Thread Priorities

In Java, you can assign priorities to threads to influence their execution order. The priority is an integer value, ranging from Thread.MIN_PRIORITY (1) to Thread.MAX_PRIORITY (10), with a default priority of Thread.NORM_PRIORITY (5). However, the thread scheduler is free to ignore these priorities.

Example: Setting Thread Priorities

```
class MyThread extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName() + " is running.");
    }
}


public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();


        t1.setPriority(Thread.MAX_PRIORITY);  // Set highest priority
        t2.setPriority(Thread.MIN_PRIORITY);  // Set lowest priority


        t1.start();
        t2.start();
    }
}
```

- The thread with the **highest priority** may be executed first, but the order of execution is determined by the thread scheduler.

---

4. Synchronization

Synchronization in Java is used to prevent multiple threads from simultaneously accessing shared resources, which could lead to inconsistent data or application errors.

Java provides the synchronized keyword to achieve thread safety. You can synchronize methods or code blocks.

Example: Synchronization Using Methods

```
class Counter {
    private int count = 0;

    // Synchronized method to ensure only one thread can modify the count at a time
    synchronized void increment() {
        count++;
        System.out.println(Thread.currentThread().getName() + ": " + count);
    }
}

public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();

        // Two threads that will access the same Counter object
        Thread t1 = new Thread(() -> counter.increment());
        Thread t2 = new Thread(() -> counter.increment());

        t1.start();
        t2.start();
    }
}
```

- The increment() method is synchronized, meaning only one thread can execute it at a time, avoiding data inconsistencies.

---

5. Inter-Thread Communication

Java provides mechanisms for inter-thread communication using wait(), notify(), and notifyAll() methods, which are defined in the Object class. These methods are used for thread coordination, where one thread waits for some condition to be met, and another thread notifies it when the condition is met.

Example: Inter-Thread Communication

```
class SharedResource {
    private int data = 0;

    public synchronized void produceData() throws InterruptedException {
        while (data >= 1) {
            wait();  // Wait if data is already produced
        }
        data++;
        System.out.println("Data produced: " + data);
        notify();  // Notify the consumer
    }

    public synchronized void consumeData() throws InterruptedException {
        while (data <= 0) {
            wait();  // Wait if no data is available to consume
        }
        data--;
        System.out.println("Data consumed: " + data);
        notify();  // Notify the producer
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        SharedResource resource = new SharedResource();

        Thread producer = new Thread(() -> {
            try {
```

```
            resource.produceData();
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    });

    Thread consumer = new Thread(() -> {
        try {
            resource.consumeData();
        } catch (InterruptedException e) {
            System.out.println(e);
        }
    });

    producer.start();
    consumer.start();

    producer.join();
    consumer.join();
  }
}
```

- The produceData() method waits if the data has already been produced, and consumeData() waits if no data is available.
- Both methods use wait() to pause execution and notify() to signal the other thread.

---

6. Deadlock

Deadlock is a situation in multithreading where two or more threads are blocked forever because they are waiting for each other to release resources. Deadlock occurs when:

- Thread 1 holds Resource A and waits for Resource B.

- Thread 2 holds Resource B and waits for Resource A.

This causes a circular dependency, preventing any thread from proceeding.

Example: Deadlock Example

```
class ResourceA {
    synchronized void methodA(ResourceB resourceB) {
        System.out.println("Thread 1: Locked Resource A, waiting for Resource B");
        resourceB.last();
    }

    synchronized void last() {
        System.out.println("Thread 1: Executing last method of Resource A");
    }
}

class ResourceB {
    synchronized void methodB(ResourceA resourceA) {
        System.out.println("Thread 2: Locked Resource B, waiting for Resource A");
        resourceA.last();
    }

    synchronized void last() {
        System.out.println("Thread 2: Executing last method of Resource B");
    }
}

public class Main {
    public static void main(String[] args) {
        ResourceA resourceA = new ResourceA();
        ResourceB resourceB = new ResourceB();
        Thread t1 = new Thread(() -> {
            resourceA.methodA(resourceB);
        });
        Thread t2 = new Thread(() -> {
            resourceB.methodB(resourceA);
        });

        t1.start();
        t2.start();
    }
}
```

- **Thread 1** locks **Resource A** and waits for **Resource B**, while **Thread 2** locks **Resource B** and waits for **Resource A**, causing a deadlock.

Avoiding Deadlock

- Avoid nested locks (lock ordering).
- Use **timeout** to detect and resolve deadlocks.

Conclusion

Multithreading is an essential concept for building efficient, responsive, and concurrent applications in Java. Key points include:

- **Thread creation** can be done by extending the Thread class or implementing the Runnable interface.
- Threads can execute concurrently, and their **priority** can be set to influence execution order.
- **Synchronization** ensures that shared resources are accessed safely by multiple threads.
- **Inter-thread communication** enables threads to wait for conditions to be met before proceeding.
- **Deadlock** is a critical issue in multithreading and can be avoided by careful resource management and design.

By understanding and applying these concepts, you can create high-performance, reliable, and responsive applications in Java.

Topic: I/O and Streams

I/O and Streams in Java (2 hours)

The Java I/O (Input/Output) system provides a powerful and flexible mechanism for interacting with external systems such as files, network sockets, and other data sources. The Java java.io package is the core library used for I/O operations, and it provides several classes to handle input and output through streams.

In Java, everything is considered as a **stream** of data. A stream is a sequence of data elements that can be read from or written to, and it can be either **byte-based** or **character-based**.

This note covers key concepts of **streams**, including reading/writing console input/output, file handling, and the concept of **serialization** in Java.

---

1. The java.io Package

The java.io package provides various classes for system input and output through data streams, serialization, file handling, etc. It contains essential classes such
as File, InputStream, OutputStream, Reader, Writer, and more.

2. Files and Directories

Java provides the File class to represent files and directories in a file system. The File class provides various methods for creating, deleting, and checking file properties.

Example: Working with Files and Directories

```
import java.io.File;

import java.io.IOException;


public class FileExample {

    public static void main(String[] args) {

        // Create a new file object representing a file or directory

        File file = new File("example.txt");


        // Check if the file exists

        if (file.exists()) {

            System.out.println("File exists: " + file.getName());

        } else {
            System.out.println("File does not exist.");
        }

        // Create a new directory
        File directory = new File("myDirectory");
        if (!directory.exists()) {
            directory.mkdir();  // Create directory
            System.out.println("Directory created.");
        }
    }
}
```

- The File class provides methods like exists(), mkdir(), delete(), and getName() for working with files and directories.
- **mkdir()** creates a directory, while **exists()** checks whether the file or directory exists.

---

3. Streams in Java

A **stream** is an abstraction used to handle I/O operations. There are two primary types of streams:

- **Byte Streams**: Used for binary data (e.g., files, images, etc.). These streams handle input and output in terms of bytes (8 bits).
- **Character Streams**: Used for character data (text files). They handle input and output in terms of characters (16 bits).

A. Byte Streams:

Byte streams are used for handling raw binary data. They provide classes like InputStream (for reading data) and OutputStream (for writing data).

- **Example classes**: FileInputStream, FileOutputStream

B. Character Streams:

Character streams are designed to handle **text** data, automatically converting characters to bytes using a specific charset (e.g., UTF-8).

- **Example classes**: FileReader, FileWriter

---

4. Reading and Writing Console Input/Output

Java provides simple mechanisms for reading input from the console and writing output to the console.

A. Reading from Console:

You can use the Scanner class or BufferedReader for reading user input.

Example: Reading from the Console Using Scanner

```
import java.util.Scanner;


public class ConsoleInput {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);


      System.out.println("Enter your name: ");
      String name = scanner.nextLine();


      System.out.println("Hello, " + name);
   }
}
```

- The Scanner class is commonly used for reading primitive types like int, float, and String from the console.

B. Writing to Console:

You can use System.out.print() or System.out.println() to display output on the console.

Example: Writing to the Console

```
public class ConsoleOutput {
   public static void main(String[] args) {
      System.out.println("Hello, World!");
   }
}
```

- The println() method prints the message and moves to the next line, while print() just prints the message on the same line.

---

5. Reading and Writing Files

Reading from and writing to files are common operations in Java. Using byte or character streams, you can read and write files.

A. Reading from a File (Character Stream)

```
import java.io.FileReader;

import java.io.BufferedReader;

import java.io.IOException;


public class FileReaderExample {

  public static void main(String[] args) {

    try (BufferedReader br = new BufferedReader(new FileReader("example.txt"))) {

      String line;

      while ((line = br.readLine()) != null) {

        System.out.println(line);  // Print the content of the file line by line

      }

    } catch (IOException e) {

      e.printStackTrace();

    }

  }

}
```

- BufferedReader is used for efficient reading of text files. The readLine() method reads one line of text at a time.
- FileReader is used to read file content as characters.

B. Writing to a File (Character Stream)

```
import java.io.FileWriter;

import java.io.BufferedWriter;

import java.io.IOException;
```

```
public class FileWriterExample {
    public static void main(String[] args) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"))) {
            writer.write("Hello, File Writer!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- FileWriter is used to write characters to a file, and BufferedWriter ensures efficient writing.

---

6. Serialization and Deserialization

Serialization is the process of converting an object into a byte stream for storage or transmission, while **deserialization** is the reverse process of converting a byte stream back into an object.

A. Serialization Interface

In Java, the Serializable interface is used to mark a class whose objects can be serialized.

B. Example: Serialization

```
import java.io.*;

class Person implements Serializable {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```
public class SerializationExample {
   public static void main(String[] args) {
      Person person = new Person("Alice", 30);

      try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("person.ser"))) {
         oos.writeObject(person);  // Serialize the object
         System.out.println("Object serialized!");
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

- The Person class implements Serializable, which makes it eligible for serialization.
- **ObjectOutputStream** writes the object to a file.

C. Deserialization Example

```
import java.io.*;

class Person implements Serializable {
   String name;
   int age;

   Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
}

public class DeserializationExample {
   public static void main(String[] args) {
      try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("person.ser"))) {
          Person person = (Person) ois.readObject();  // Deserialize the object
          System.out.println("Object deserialized!");
          System.out.println("Name: " + person.name + ", Age: " + person.age);
      } catch (IOException | ClassNotFoundException e) {
          e.printStackTrace();
      }
   }
}
```

- **ObjectInputStream** reads the object from the file and restores it to its original state.
- The deserialized object can then be used in the program.

Conclusion

Java I/O is a crucial part of programming that enables communication between your application and external systems. The key concepts include:

- **File Handling** with File, including creating and checking files/directories.
- **Streams** for reading and writing data, with **byte streams** (e.g., FileInputStream, FileOutputStream) and **character streams** (e.g., FileReader, FileWriter).
- **Console Input/Output** using classes like Scanner and System.out.
- **Serialization** and **Deserialization** for converting objects to byte streams and vice versa, using Serializable interface.

These concepts allow you to interact with files, handle user input, and ensure data is saved and restored properly in Java applications.

Topic: Understanding Core Packages

Understanding Core Packages in Java (3 hours)

Java provides a rich set of core packages that contain classes and methods to help with everyday tasks such as mathematical computations, data manipulation, random number generation, and much more. These packages form the foundation of Java programming, allowing developers to build robust applications efficiently.

This note will cover two essential core Java packages:

1. **java.lang Package**: Includes fundamental classes such as Math, and wrapper classes like Integer, Double, Character, etc.
2. **java.util Package**: Provides utility classes such as Vector, Stack, Hashtable, Random, and others.

---

1. java.lang Package

The java.lang package is automatically imported into every Java program and contains fundamental classes that are essential for Java programming. These include the Math class for mathematical operations and **wrapper classes** for converting primitive data types to objects.

A. java.lang.Math Class

The Math class provides a collection of static methods for performing common mathematical operations such as basic arithmetic, trigonometry, exponentiation, and more.

Common Methods in Math Class

- **Math.abs(x)**: Returns the absolute value of x.
- **Math.max(x, y)**: Returns the larger of the two numbers x and y.
- **Math.min(x, y)**: Returns the smaller of the two numbers x and y.
- **Math.pow(x, y)**: Returns x raised to the power of y.
- **Math.sqrt(x)**: Returns the square root of x.
- **Math.random()**: Generates a random number between 0.0 (inclusive) and 1.0 (exclusive).

Example: Using Math Class

```
public class MathExample {

  public static void main(String[] args) {

    System.out.println("Absolute value of -10: " + Math.abs(-10));

    System.out.println("Maximum of 10 and 20: " + Math.max(10, 20));

    System.out.println("Square root of 16: " + Math.sqrt(16));

    System.out.println("Random number: " + Math.random());

  }

}
```

Output:

```
Absolute value of -10: 10

Maximum of 10 and 20: 20
```

```
Square root of 16: 4.0

Random number: 0.9178471480345451
```

B. Wrapper Classes

Java provides **wrapper classes** for each of the primitive data types. These classes wrap the primitive values into objects and provide useful methods for conversion and manipulation.

- **Integer**: Wrapper for the int type.
- **Double**: Wrapper for the double type.
- **Float**: Wrapper for the float type.
- **Byte**: Wrapper for the byte type.
- **Short**: Wrapper for the short type.
- **Long**: Wrapper for the long type.
- **Character**: Wrapper for the char type.
- **Boolean**: Wrapper for the boolean type.

Example: Using Wrapper Classes

```java
public class WrapperExample {
    public static void main(String[] args) {
        // Convert string to integer
        String str = "100";
        int num = Integer.parseInt(str);
        System.out.println("Integer: " + num);

        // Convert string to double
        String str2 = "3.14";
        double pi = Double.parseDouble(str2);
        System.out.println("Double: " + pi);

        // Autoboxing and Unboxing
        Integer i = 5;  // Autoboxing
        int n = i;      // Unboxing
        System.out.println("Autoboxed Integer: " + i + ", Unboxed int: " + n);
    }
}
```

Output:

```
Integer: 100
Double: 3.14
Autoboxed Integer: 5, Unboxed int: 5
```

Autoboxing refers to automatically converting a primitive type to its corresponding wrapper class, and **unboxing** refers to converting a wrapper class back into its primitive form.

---

2. java.util Package

The java.util package provides several classes and interfaces that are essential for storing, processing, and manipulating data in a variety of ways. It includes utility classes for data structures, random number generation, and more.

A. Core Classes in java.util

1. **Vector**: A dynamic array that can grow as needed to accommodate new elements. It is part of the legacy collection classes but is synchronized, which makes it thread-safe.
   - **Vector Methods**: add(), remove(), get(), size(), capacity()

```
import java.util.Vector;

public class VectorExample {
    public static void main(String[] args) {
        Vector<Integer> vector = new Vector<>();
        vector.add(10);
        vector.add(20);
        vector.add(30);
        System.out.println("Vector: " + vector);
        System.out.println("Size of Vector: " + vector.size());
    }
```

13. }
14. **Stack**: A subclass of Vector that represents a stack data structure. It follows the LIFO (Last In, First Out) principle.
    - **Stack Methods**: push(), pop(), peek(), isEmpty()
15. import java.util.Stack;
16.
17. public class StackExample {
18.   public static void main(String[] args) {
19.   Stack<Integer> stack = new Stack<>();
20.   stack.push(10);
21.   stack.push(20);
22.   stack.push(30);
23.   System.out.println("Top of stack: " + stack.peek());
24.   System.out.println("Popped element: " + stack.pop());
25.   System.out.println("Stack after pop: " + stack);
26.   }
27. }
28. **Dictionary** (Legacy): A key-value data structure used for storing data. It is the parent class of Hashtable.
    - **Dictionary Methods**: get(), put(), remove()
29. **Hashtable**: A hash table that implements the Map interface. It is synchronized and stores key-value pairs, allowing for fast retrieval of data.
30. import java.util.Hashtable;
31.
32. public class HashtableExample {
33.   public static void main(String[] args) {
34.   Hashtable<String, Integer> hashtable = new Hashtable<>();
35.   hashtable.put("Apple", 100);
36.   hashtable.put("Banana", 50);
37.   System.out.println("Value for 'Apple': " + hashtable.get("Apple"));

```
    }
}
```

B. Enumerations

An **enumeration** represents a collection of constant values. Java provides the Enumeration interface, which has methods like hasMoreElements() and nextElement() to iterate over collections.

```
import java.util.Vector;
import java.util.Enumeration;

public class EnumerationExample {
  public static void main(String[] args) {
    Vector<String> vector = new Vector<>();
    vector.add("Java");
    vector.add("Python");
    vector.add("C++");

    Enumeration<String> enumeration = vector.elements();
    while (enumeration.hasMoreElements()) {
      System.out.println(enumeration.nextElement());
    }
  }
}
```

C. Random Number Generation

The Random class in the java.util package is used for generating pseudo-random numbers.

- **nextInt()**: Generates a random integer.
- **nextDouble()**: Generates a random double between 0.0 (inclusive) and 1.0 (exclusive).
- **nextBoolean()**: Generates a random boolean value.

Example: Random Number Generation

```
import java.util.Random;
```

```
public class RandomExample {
   public static void main(String[] args) {
      Random random = new Random();

      int randInt = random.nextInt(100);  // Random integer between 0 and 99
      double randDouble = random.nextDouble();  // Random double between 0.0 and 1.0
      boolean randBoolean = random.nextBoolean();  // Random boolean value

      System.out.println("Random Integer: " + randInt);
      System.out.println("Random Double: " + randDouble);
      System.out.println("Random Boolean: " + randBoolean);
   }
}
```

Conclusion

The core packages java.lang and java.util provide a variety of fundamental tools and utilities that every Java programmer must understand and use regularly:

- The **java.lang** package offers basic utilities like the Math class for mathematical operations and **wrapper classes** for primitive types, which help in object-oriented programming.
- The **java.util** package provides powerful data structures such as Vector, Stack, Hashtable, and random number generation utilities like the Random class.

Understanding and leveraging these core packages will make it easier to develop efficient and well-structured Java applications.

Topic: Holding Collection of Data

Holding Collection of Data in Java (3 hours)

Java provides a variety of classes and interfaces for holding and manipulating collections of data. The **Collection Framework** is a set of interfaces and classes that allow you to handle data in a structured and efficient manner. This note will cover the concepts of **arrays**, **collections**, and the main **Map**, **List**, and **Set** interfaces along with their implementations, and how you can use them effectively.

---

1. Arrays in Java

An **array** is a fixed-size, homogeneous data structure that holds multiple values of the same type. It is a basic way to store data in Java.

A. Declaring and Initializing Arrays

You can declare and initialize arrays as follows:

```
public class ArrayExample {
  public static void main(String[] args) {
    // Declaring and initializing an array
    int[] numbers = new int[5];  // Array of integers with size 5
    numbers[0] = 10;
    numbers[1] = 20;
    numbers[2] = 30;
    numbers[3] = 40;
    numbers[4] = 50;

    // Printing array elements
    for (int num : numbers) {
      System.out.println(num);
    }
  }
}
```

- Arrays are indexed starting from 0, and their size is fixed after initialization.
- You can access an element using its index.

---

2. The Collection Framework

The **Collection Framework** provides a unified architecture for handling different types of data structures. It includes classes for **lists**, **sets**, **maps**, and **queues**, as well as interfaces like Collection, List, Set, and Map.

A. The Collection Interface

The Collection interface is the root interface in the collection hierarchy. It represents a group of objects and provides basic methods like add(), remove(), size(), and clear().

B. The List Interface

The List interface represents an ordered collection of elements (also known as a sequence). Lists allow duplicate elements and provide methods for positional access to elements.

- Common implementations of the List interface:
  - **ArrayList**: A resizable array-based implementation.
  - **LinkedList**: A doubly-linked list-based implementation.

Example: Using ArrayList

```
import java.util.ArrayList;

public class ListExample {
  public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<>();
    list.add("Apple");
    list.add("Banana");
    list.add("Cherry");

    System.out.println("List: " + list);
    System.out.println("Size of the list: " + list.size());

    // Accessing elements
    System.out.println("First element: " + list.get(0));
  }
}
```

Example: Using LinkedList

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");

        System.out.println("LinkedList: " + list);
    }
}
```

- **ArrayList** offers fast random access and is generally preferred when the list is frequently accessed by index.
- **LinkedList** is more efficient when there are frequent insertions or deletions in the middle of the list.

C. The Set Interface

The Set interface represents a collection that does not allow duplicate elements. It models the mathematical set abstraction, where no two elements can be the same.

- Common implementations of the Set interface:
  - **HashSet**: Implements a set backed by a hash table, which makes it very fast for checking membership.
  - **TreeSet**: Implements a set backed by a tree structure, which maintains elements in sorted order.

Example: Using HashSet

```
import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();
        set.add("Apple");
        set.add("Banana");
        set.add("Apple");  // Duplicate element will be ignored
        System.out.println("HashSet: " + set);
    }
}
```

Example: Using TreeSet

```
import java.util.TreeSet;

public class TreeSetExample {
    public static void main(String[] args) {
        TreeSet<String> set = new TreeSet<>();
        set.add("Banana");
        set.add("Apple");
        set.add("Cherry");

        System.out.println("TreeSet (Sorted Order): " + set);
    }
}
```

- **HashSet** provides constant time performance for basic operations like add(), remove(), and contains(), but it does not maintain any order of elements.
- **TreeSet** provides a sorted order of elements, but operations take longer than HashSet due to the tree structure.

D. The Map Interface

The Map interface represents a collection of key-value pairs, where each key is unique. A Map does not extend the Collection interface but is part of the collection framework.

- Common implementations of the Map interface:
    - **HashMap**: A hash table-based implementation that allows fast lookups.
    - **TreeMap**: A red-black tree-based implementation that keeps the keys in sorted order.

Example: Using HashMap

```
import java.util.HashMap;
```

```
public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("Apple", 100);
        map.put("Banana", 50);
        map.put("Cherry", 75);

        System.out.println("Map: " + map);
        System.out.println("Price of Banana: " + map.get("Banana"));
    }
}
```

- **HashMap** provides constant time complexity for get() and put() operations, but the order of elements is not guaranteed.
- **TreeMap** maintains the order of the keys, and the keys are sorted in natural order or by a custom comparator.

---

3. Iterating Over Collections

In Java, you can use **Iterators** to traverse through elements in a collection. An **Iterator** is an object that allows you to loop through the collection's elements one by one.

A. Using an Iterator

An Iterator is used to iterate over a collection, and it provides methods like hasNext(), next(), and remove().

Example: Using Iterator

```
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");

        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

- **hasNext()** checks if there are more elements to iterate through.
- **next()** returns the next element.
- **remove()** removes the current element from the collection.

---

4. Comparator and Comparable Interfaces

Both the **Comparator** and **Comparable** interfaces are used for sorting elements in a collection.

A. Comparable Interface

The Comparable interface is used when you want objects to be compared naturally (e.g., sorting strings alphabetically or numbers numerically). The compareTo() method is used to define the natural ordering of objects.

Example: Using Comparable

```
import java.util.ArrayList;
import java.util.Collections;

class Person implements Comparable<Person> {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int compareTo(Person other) {
        return this.age - other.age;  // Sort by age in ascending order
    }

    @Override
    public String toString() {
        return name + ": " + age;
    }
}

public class ComparableExample {
    public static void main(String[] args) {
        ArrayList<Person> list = new ArrayList<>();
        list.add(new Person("Alice", 30));
        list.add(new Person("Bob", 25));
        list.add(new Person("Charlie", 35));

        Collections.sort(list);  // Sorting using compareTo() method
        System.out.println(list);
    }
}
```

B. Comparator Interface

The Comparator interface is used to define custom sorting logic, especially when you want to compare objects based on different attributes (e.g., sorting by name or age).

Example: Using Comparator

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return name + ": " + age;
    }
}

class SortByName implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.name.compareTo(p2.name);  // Sort by name alphabetically
    }
}

public class ComparatorExample {
    public static void main(String[] args) {
        ArrayList<Person> list = new ArrayList<>();
        list.add(new Person("Alice", 30));
        list.add(new Person("Bob", 25));
        list.add(new Person("Charlie", 35));

        Collections.sort(list, new SortByName());  // Sorting using Comparator
        System.out.println(list);
    }
}
```

Conclusion

Java provides powerful and flexible ways to hold and manipulate collections of data through the **Collection Framework**. Key points include:

1. **Arrays**: Fixed-size data structures for storing elements.
2. **Collection Interfaces**:
    - **List**: Ordered collections (e.g., ArrayList, LinkedList).
    - **Set**: Unordered collections with no duplicates (e.g., HashSet, TreeSet).
    - **Map**: Key-value pair collections (e.g., HashMap, TreeMap).
3. **Iterators**: Provide a consistent way to iterate over collections.
4. **Sorting**: Using **Comparable** and **Comparator** for sorting collections.

Understanding and utilizing these collections effectively will allow you to create efficient and well-structured Java programs.

Topic:Java Applications

Java Applications: Building GUI Applications with AWT & Swing (8 hours)

Java provides two powerful libraries for building graphical user interfaces (GUIs): **AWT (Abstract Window Toolkit)** and **Swing**. Both libraries offer a set of components and tools that allow you to create rich, interactive user interfaces. This note will introduce you to the basics of **AWT** and **Swing**, the components used to build interfaces, and the principles of event handling, layout management, and more advanced features like MDI (Multiple Document Interface).

---

1. Introduction to AWT and Swing

- **AWT (Abstract Window Toolkit)**: Introduced in Java 1.0, AWT is a set of APIs used for building graphical user interfaces. It provides basic GUI components like buttons, text fields, and labels. However, AWT components are heavy-weight, meaning they rely on the underlying operating system for rendering, which can result in less flexibility and inconsistent look-and-feel across platforms.
- **Swing**: Swing, an extension of AWT, was introduced later and provides more sophisticated GUI components. Swing components are **lightweight**, meaning they are drawn entirely by Java, giving them a consistent look-and-feel across platforms. Swing also supports richer controls like tables, trees, and text areas.

---

2. JFrame – A Top-Level Window in Swing

A **JFrame** is a top-level container in Swing that represents a window on the screen. It is part of the Swing library and serves as the main window in a GUI application.

Creating a JFrame:

```
import javax.swing.JFrame;


public class SimpleWindow {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Simple Swing Window");
    frame.setSize(400, 300);  // Set the size of the window
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  // Close the application on window close
    frame.setVisible(true);  // Display the window
  }
}
```

- setSize(width, height): Specifies the window's size.
- setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE): Defines the action when the window is closed (exit the program).
- setVisible(true): Makes the window visible.

---

3. Swing Components

Swing provides a wide range of components for building user interfaces. Below are some of the commonly used Swing components:

A. JLabel

A JLabel is a non-editable text component that can display text or an image.

```
import javax.swing.*;


public class JLabelExample {
   public static void main(String[] args) {
      JFrame frame = new JFrame("JLabel Example");
      JLabel label = new JLabel("Hello, Swing!", JLabel.CENTER);
      frame.add(label);
      frame.setSize(300, 200);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

B. JTextField

A JTextField is a text input field where users can enter a single line of text.

```
import javax.swing.*;


public class JTextFieldExample {
   public static void main(String[] args) {
      JFrame frame = new JFrame("JTextField Example");
      JTextField textField = new JTextField(20);
      frame.add(textField);
      frame.setSize(300, 200);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

C. JButton

A JButton is a clickable button that can trigger actions when clicked.

```
import javax.swing.*;

import java.awt.event.*;


public class JButtonExample {

   public static void main(String[] args) {

      JFrame frame = new JFrame("JButton Example");

      JButton button = new JButton("Click Me!");

      button.addActionListener(new ActionListener() {

         public void actionPerformed(ActionEvent e) {

            JOptionPane.showMessageDialog(frame, "Button clicked!");

         }

      });

      frame.add(button);

      frame.setSize(300, 200);

      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.setVisible(true);

   }

}
```

---

4. Event Handling in Swing Applications

Event handling is a key concept in GUI programming. In Swing, you handle events like button clicks, mouse movements, and key presses using listeners and listener interfaces.

A. Event Listener Example: JButton Click

In the above example, the JButton uses an ActionListener to detect clicks.

```
button.addActionListener(new ActionListener() {

   public void actionPerformed(ActionEvent e) {

      // Handle button click event

   }
});
```

- ActionListener: Responds to action events such as button clicks.
- MouseListener: Responds to mouse events (e.g., mouse clicks, mouse movements).
- KeyListener: Responds to keyboard events.

---

5. Layout Management

Swing provides different layout managers for organizing components in a container. The layout managers handle the positioning of components inside a container automatically.

A. FlowLayout (Default Layout Manager for JFrame)

- Components are arranged in the order they are added, left to right.

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

public class FlowLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("FlowLayout Example");
        frame.setLayout(new FlowLayout());  // Set FlowLayout manager

        JButton button1 = new JButton("Button 1");
        JButton button2 = new JButton("Button 2");
        JButton button3 = new JButton("Button 3");

        frame.add(button1);
        frame.add(button2);
        frame.add(button3);

        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

B. BorderLayout

- The container is divided into five areas: North, South, East, West, and Center.

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

public class BorderLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("BorderLayout Example");
        frame.setLayout(new BorderLayout());

        JButton northButton = new JButton("North");
        JButton southButton = new JButton("South");
        JButton centerButton = new JButton("Center");

        frame.add(northButton, BorderLayout.NORTH);
        frame.add(southButton, BorderLayout.SOUTH);
        frame.add(centerButton, BorderLayout.CENTER);

        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

C. GridLayout

- Divides the container into a grid of rows and columns.

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

```
public class GridLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("GridLayout Example");
        frame.setLayout(new GridLayout(2, 2));  // 2 rows, 2 columns

        JButton button1 = new JButton("Button 1");
        JButton button2 = new JButton("Button 2");
        JButton button3 = new JButton("Button 3");
        JButton button4 = new JButton("Button 4");

        frame.add(button1);
        frame.add(button2);
        frame.add(button3);
        frame.add(button4);

        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

---

6. Choice Components

A. JCheckBox

A JCheckBox allows the user to choose one or more options from a set.

```
import javax.swing.*;

public class JCheckBoxExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JCheckBox Example");
        JCheckBox checkBox = new JCheckBox("Accept Terms and Conditions");

        frame.add(checkBox);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

B. JRadioButton

A JRadioButton allows the user to choose one option from a set of radio buttons. They are typically used in groups where only one button can be selected.

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

public class JRadioButtonExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JRadioButton Example");
        JRadioButton radio1 = new JRadioButton("Option 1");
        JRadioButton radio2 = new JRadioButton("Option 2");

        ButtonGroup group = new ButtonGroup();  // Ensures only one radio button is selected
        group.add(radio1);
        group.add(radio2);

        frame.setLayout(new FlowLayout());
        frame.add(radio1);
        frame.add(radio2);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

---

7. Menus in Swing

You can create menus using JMenuBar, JMenu, and JMenuItem.

```
import javax.swing.*;

public class MenuExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Menu Example");

        JMenuBar menuBar = new JMenuBar();

        JMenu fileMenu = new JMenu("File");
        JMenuItem openItem = new JMenuItem("Open");
        fileMenu.add(openItem);

        JMenu editMenu = new JMenu("Edit");
        JMenuItem cutItem = new JMenuItem("Cut");
        editMenu.add(cutItem);

        menuBar.add(fileMenu);
        menuBar.add(editMenu);

        frame.setJMenuBar(menuBar);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

---

8. JTable for Displaying Data

JTable is used to display data in a tabular form.

```
import javax.swing.*;
import javax.swing.table.DefaultTableModel;

public class JTableExample {
    public static void main(String[] args) {
        String[] columns = {"Name", "Age", "City"};
        Object[][] data = {
            {"Alice", 25, "New York"},
            {"Bob", 30, "Los Angeles"},
            {"Charlie", 35, "Chicago"}
        };

        JTable table = new JTable(data, columns);
        JScrollPane scrollPane = new JScrollPane(table);

        JFrame frame = new JFrame("JTable Example");
        frame.add(scrollPane);
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

9. MDI (Multiple Document Interface) using JDesktopPane

MDI applications allow multiple windows to open inside a parent window. Use JDesktopPane to create MDI applications.

```
import javax.swing.*;


public class MDIExample {
  public static void main(String[] args) {
    JFrame frame = new JFrame("MDI Example");
    JDesktopPane desktopPane = new JDesktopPane();


    JInternalFrame internalFrame = new JInternalFrame("Internal Frame", true, true, true, true);
    internalFrame.setSize(200, 150);
    internalFrame.setVisible(true);


    desktopPane.add(internalFrame);
    frame.add(desktopPane);
    frame.setSize(400, 300);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

10. Using IDEs like NetBeans & JBuilder

- **NetBeans** and **JBuilder** are popular IDEs for building Java applications with GUI. These IDEs allow you to design your GUI using drag-and-drop functionality, saving development time.
- **NetBeans** provides tools for Swing-based development and integrates visual components with Java code seamlessly.

Conclusion

In Java, building GUI applications is facilitated by **AWT** and **Swing**. Key components like **JFrame**, **JLabel**, **JButton**, and layout managers such as **FlowLayout**, **BorderLayout**, and **GridLayout** help create interactive and well-organized interfaces. Event handling is crucial for responding to user actions, and advanced features like **JTable**, **JList**, and **JComboBox** enable you to build rich, data-driven applications. Additionally, tools like **NetBeans** and **JBuilder** streamline the development process with drag-and-drop functionality for creating complex UIs.

Topic:Introduction to Java Applets

Introduction to Java Applets (1 Hour)

Java applets are small programs written in Java that run inside a web browser. Historically, applets were widely used to create interactive applications on web pages. However, due to security issues and the rise of other technologies (like JavaScript, HTML5, and CSS), applets are now largely deprecated and are no longer supported by most modern web browsers. Despite this, understanding applets is important for historical context and maintaining legacy systems.

This note provides an overview of Java applets, their lifecycle, and how to build simple applets with basic controls and animations.

---

1. Definition of Java Applets

An **applet** is a small Java application designed to run within a web browser or an applet viewer. Unlike stand-alone Java applications that are executed directly by the Java Virtual Machine (JVM), applets are embedded into HTML pages and executed by a Java-enabled browser.

Applets are typically used for interactive content like games, simulations, and dynamic data displays. Applet programs must adhere to specific life cycle methods that govern their behavior in a browser environment.

---

2. Applet Lifecycle Methods

The lifecycle of a Java applet is controlled by several methods that are automatically invoked by the browser or applet viewer. These methods manage the applet's initialization, execution, and termination. The key lifecycle methods are:

- **init()**: This method is called when the applet is first loaded into memory. It is used to initialize variables, load resources, and set up the applet's environment.
- public void init() {
-     // Initialization code (e.g., setting up resources)
- }

- **start()**: This method is invoked after init() and every time the applet is revisited (e.g., when the user navigates back to the page containing the applet). It is used to start animation or other ongoing tasks.
- public void start() {
-     // Start tasks like animations or threads
- }
- **paint(Graphics g)**: This method is called whenever the applet's display needs to be redrawn, such as after a resize or repaint request. It is where graphical output is rendered.
- public void paint(Graphics g) {
-     g.drawString("Hello, Applet!", 20, 20); // Example of rendering text
- }
- **stop()**: This method is called when the applet is no longer visible, either when the user navigates away from the page or when the applet is explicitly stopped. It is used to stop animations or release resources.
- public void stop() {
-     // Stop any ongoing tasks or release resources
- }
- **destroy()**: This method is invoked when the applet is unloaded. It is used to release any remaining resources or perform cleanup before the applet is destroyed.
- public void destroy() {
-     // Clean-up tasks like closing files or releasing resources
- }

---

3. Building a Simple Applet

Let's now build a simple Java applet that displays a message in a browser or applet viewer. The applet will use basic methods like init(), start(), and paint().

```
import java.applet.Applet;

import java.awt.Graphics;


public class SimpleApplet extends Applet {

  // Override init method
```

```
    public void init() {
        setBackground(java.awt.Color.white);  // Set background color
    }


    // Override paint method to display text
    public void paint(Graphics g) {
        g.drawString("Hello, this is a simple applet!", 50, 50);
    }
}
```

- The SimpleApplet extends the Applet class.
- The init() method initializes the applet, such as setting the background color.
- The paint() method is responsible for rendering the text on the applet's window.

To view this applet, you would typically embed it into an HTML file using the <applet> tag. However, most modern browsers no longer support applets.

Example HTML code to display the applet (used historically):

```
<applet code="SimpleApplet.class" width="300" height="200">
</applet>
```

---

4. Using Applet Viewer

AppletViewer is a tool that allows you to run and debug Java applets without needing a browser. It is part of the Java Development Kit (JDK).

To use AppletViewer:

1. Compile the applet code:
2. javac SimpleApplet.java
3. Create an HTML file that references the applet (e.g., SimpleApplet.html):
4. <html>
5.   <body>
6.   <applet code="SimpleApplet.class" width="300" height="200">
7.   </applet>

8. </body>

9. </html>

10. Use AppletViewer to run the applet:

11. appletviewer SimpleApplet.html

This will open a window displaying the applet without the need for a browser.

---

5. Adding Controls: Animation Concepts

Java applets can be interactive, allowing you to add controls like buttons, sliders, and other UI elements. Animation can also be implemented using threads and the paint() method.

Example: Simple Animation in Applet

```
import java.applet.Applet;

import java.awt.Color;

import java.awt.Graphics;


public class AnimationApplet extends Applet implements Runnable {

  int x = 10; // X coordinate of the moving object

  Thread animationThread;


  public void init() {

    setBackground(Color.white);  // Set background color

    animationThread = new Thread(this);

    animationThread.start();  // Start the animation thread

  }


  public void paint(Graphics g) {

    g.setColor(Color.blue);

    g.fillRect(x, 50, 50, 50);  // Draw a rectangle that moves

  }
```

```
    public void run() {
        while (true) {
            x += 5;  // Move the rectangle horizontally
            if (x > getWidth()) {
                x = 0;  // Reset the position when it goes off screen
            }
            repaint();  // Repaint the applet to update the movement
            try {
                Thread.sleep(50);  // Pause to slow down the animation
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public void stop() {
        animationThread.stop();  // Stop the animation when the applet is stopped
    }
}
```

In this example:

- The AnimationApplet extends the Applet class and implements the Runnable interface to create an animation.
- A Thread object is used to create a new thread for the animation, and the object moves horizontally across the screen.
- The repaint() method is called in the run() method to continuously redraw the moving object.

---

6. Summary

- **Definition**: Java applets are small Java programs that run in a web browser or applet viewer.
- **Applet Lifecycle**: The key methods include init(), start(), paint(), stop(), and destroy(), which control the applet's behavior at different stages.

- **Simple Applet**: A basic applet can be built by extending the Applet class and overriding lifecycle methods such as init() and paint().
- **Applet Viewer**: AppletViewer is a tool used to run and debug applets without requiring a web browser.
- **Animation in Applets**: Animation can be added to applets using threads and the paint() method to continuously update the display.

Although applets have been phased out from modern web development, understanding them is important for legacy systems and for grasping foundational Java concepts.

Topic:Database Programming using JDBC

Database Programming using JDBC (2 Hours)

Java Database Connectivity (JDBC) is a standard API for connecting Java applications to relational databases. JDBC provides a set of classes and interfaces that allow Java programs to execute SQL statements, retrieve results, and manipulate databases.

This note will focus on using the essential JDBC interfaces—Connection, Statement, and ResultSet—to manipulate data with databases, including how to set up a database connection and execute basic SQL queries.

---

1. Overview of JDBC

JDBC provides a simple, consistent interface for interacting with databases. It supports various database management systems (DBMS) like MySQL, PostgreSQL, Oracle, etc., using the same set of methods.

JDBC enables the following database operations:

- Connecting to the database
- Executing SQL queries and updates
- Processing the results of queries
- Handling exceptions and managing transactions

---

2. Setting Up JDBC

To use JDBC, you'll need to:

1. Install the relevant JDBC driver for your database (e.g., MySQL JDBC driver, PostgreSQL JDBC driver).

2. Load the driver class into your Java application.
3. Establish a connection to the database.
4. Perform the required database operations (query, update, etc.).

Example of loading the JDBC driver (for MySQL):

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

---

3. Key JDBC Interfaces

The following are the primary interfaces used in JDBC for interacting with the database:

a. Connection Interface

The Connection interface is used to establish a connection to the database. It allows you to create Statement objects, execute SQL queries, and manage transactions.

Key Methods of Connection:

- **createStatement()**: Creates a Statement object for sending SQL queries to the database.
- **prepareStatement(String sql)**: Creates a PreparedStatement object for executing parameterized SQL queries.
- **close()**: Closes the connection to the database.

Example of establishing a connection:

```
import java.sql.*;


public class JdbcExample {
  public static void main(String[] args) {
    try {
      // Load the MySQL JDBC driver
      Class.forName("com.mysql.cj.jdbc.Driver");


      // Establish connection
      Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase",
"root", "password");


      System.out.println("Connection established!");
```

```
            // Close the connection
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

---

b. Statement Interface

The Statement interface is used to send SQL queries to the database. It is commonly used for simple queries that do not require parameters. You can execute SQL queries and updates using Statement.

Key Methods of Statement:

- **executeQuery(String sql)**: Executes a SQL query and returns a ResultSet.
- **executeUpdate(String sql)**: Executes SQL update statements (INSERT, UPDATE, DELETE) and returns the number of affected rows.

Example of using Statement to query a database:

```
import java.sql.*;


public class JdbcStatementExample {
    public static void main(String[] args) {
        try {
            // Load the MySQL JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");


            // Establish connection
            Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase",
"root", "password");


            // Create a Statement object
            Statement stmt = conn.createStatement();

            // Execute a query
            ResultSet rs = stmt.executeQuery("SELECT * FROM users");

            // Process the result set
            while (rs.next()) {
                System.out.println("ID: " + rs.getInt("id"));
                System.out.println("Name: " + rs.getString("name"));
            }

            // Close the result set and statement
            rs.close();
            stmt.close();
            conn.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

In this example:

- We create a Statement object with the createStatement() method.
- We execute a SQL query to retrieve all records from the users table using executeQuery().
- We process the ResultSet to print the data.

---

c. ResultSet Interface

The ResultSet interface is used to store and iterate over the data retrieved from the database.
A ResultSet represents the result set of a query, which can be processed row by row.

Key Methods of ResultSet:

- **next()**: Moves the cursor to the next row in the result set.
- **getInt(String columnName)**: Retrieves the value of the specified column (by column name) as an integer.
- **getString(String columnName)**: Retrieves the value of the specified column (by column name) as a string.

---

4. Manipulating Data with JDBC

a. Inserting Data (Using executeUpdate)

You can use the Statement interface's executeUpdate() method to execute SQL commands such as INSERT, UPDATE, and DELETE.

Example of inserting a record into the users table:

```
import java.sql.*;


public class JdbcInsertExample {
    public static void main(String[] args) {
        try {
            // Load the MySQL JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");


            // Establish connection
            Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase",
"root", "password");


            // Create a Statement object
            Statement stmt = conn.createStatement();


            // Execute an INSERT SQL query
            String insertQuery = "INSERT INTO users (name, email) VALUES ('John Doe',
'john.doe@example.com')";
```

```
            int rowsAffected = stmt.executeUpdate(insertQuery);
            System.out.println("Rows affected: " + rowsAffected);

            // Close the statement and connection
            stmt.close();
            conn.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

In this example:

- We use executeUpdate() to insert a new record into the users table.
- The method returns the number of rows affected.

b. Updating Data (Using executeUpdate)

Similarly, you can update existing data using SQL UPDATE queries.

Example of updating a user's email:

```
String updateQuery = "UPDATE users SET email = 'new.email@example.com' WHERE id = 1";
int rowsAffected = stmt.executeUpdate(updateQuery);
```

c. Deleting Data (Using executeUpdate)

To delete records, use SQL DELETE queries.

Example of deleting a record:

```
String deleteQuery = "DELETE FROM users WHERE id = 1";
int rowsAffected = stmt.executeUpdate(deleteQuery);
```

---

5. Closing Resources

After completing database operations, always ensure that you close the Connection, Statement, and ResultSet objects to avoid memory leaks and other resource issues. This can be done in a finally block or using the try-with-resources statement introduced in Java 7.

Example of closing resources:

```
try {
   // Execute database operations
} catch (SQLException e) {
   e.printStackTrace();
} finally {
   try {
      if (rs != null) rs.close();
      if (stmt != null) stmt.close();
      if (conn != null) conn.close();
   } catch (SQLException e) {
      e.printStackTrace();
   }
}
```

---

6. Summary

- **Connection Interface**: Establishes the connection to the database.
- **Statement Interface**: Sends SQL queries and updates to the database.
- **ResultSet Interface**: Processes the results of SQL queries.
- **Execute Methods**: Use executeQuery() for SELECT queries and executeUpdate() for INSERT, UPDATE, and DELETE operations.
- **Resource Management**: Always close database resources to prevent memory leaks.

JDBC provides a flexible and powerful way to interact with relational databases from Java programs.

Topic:Syllabus

Course Title: Object Oriented Programming in Java (3 Cr.)
Course Code: CACS2O4

Year/Semester: II/III
Class Load: 6 Hrs. / Week (Theory: 3 Hrs, Tutorial: 1, Practical: 2 Hrs.)

Course Description

This course covers preliminary concepts of object-oriented approach in programming with basic skills using Java. Control structures, Classes, methods and argument passing and iteration; graphical user interface basics Programming and documentation style.

Course Objectives
The general objectives of this course are to provide fundamental concepts of Object Oriented Programming and make students familiar with the Java environment and its applications.

Course Contents

Unit 1: Introduction to Java [2 Hrs.]
Definition, History of Java, The Internet and Java's Place in IT, Applications and Applets, Java Virtual Machine, Byte Code- not an Executable code, Procedure-Oriented vs. Object-Oriented Programming, Compiling and Running a Simple Program, Setting up your Computer for Java Environment, Writing **a** Program, Compiling, Interpreting and Running the Program, Handling Common Errors

Unit 2: Tokens, Expressions and Control Structures [5 Hrs.]
Primitive Data Types: Integers, Floating-Point types, Characters, Booleans; User-Defined Data Types, Declarations, Constants, Identifiers, Literals, Type Conversion and Casting, Variables: Variable Definition and Assignment, Default Variable Initializations; Command-Line Arguments, Arrays of Primitive Data Types, Comment Syntax, Garbage Collection, Expressions, Using Operators: Arithmetic, Bitwise, Relational, Logical, Assignment, Conditional, Shift, Ternary, Auto-increment and Auto-decrement; Using Control Statements(Branching: if, switch; Looping: while, do-while, for; Jumping statements: break, continue and return)

Unit 3: Object Oriented Programming Concepts [9 Hrs.]
Fundamentals of Classes: A Simple Class, Creating Class Instances, Adding methods to a class, Calling Functions/Methods; Abstraction. Encapsulation. Using 'this' keyword, ConstrucCors -115efault constructors, Parameterized constructors, More on methods: Passing by' Value, by Reference, Access Control. Methods that Return Values, Polymorphism and Method Overloading, Recursion; Nested and Inner Classes

Unit 4: Inheritance & Packaging [3 Hrs.]
inheritance: Using 'extends' keyword, Subclasses and Superclasses, 'super' keyword usage, Overriding Methods, Dynamic Method Dispatch; The Object class, Abstract and Final Classes, Packages: Defining a Package, Importing a Package: Access Control; Interfaces: Defining an Interface, Implementing and applying interfaces.

Unit 5: Handling Error/Exception [2 Hrs.]
Basic Exceptions, Proper use of exceptions, User defined Exceptions, Catching Exception: try, catch; Throwing and rethrowing: throw, throws; Cleaning up using the finally clause.

Unit 6: Handling Strings [2 Hrs.]
Creation, Concatenation and Conversion of a String, Changing Case, Character Extraction, String Comparison, Searching Strings, Modifying Strings, String Buffer.

Unit 7: Threads [3 Hrs.]
Create/Instantiate/Start New Threads: Extending java.lang.Thread, Implementing java.lang.Runnable Interface; Understand Thread Execution, Thread Priorities, Synchronization, Inter-Thread Communication, Deadlock

Unit 8: I/O and Streams [2 Hrs.]
java.io package, Files and directories, Streams: Byte Streams and Character Streams; Reading/Writing Console Input/Output, Reading and Writing files, The Serialization Interface, Serialization & Deserialization.

Unit 9: Understanding Core Packages [3 Hrs.]
Using java.lang Package: java.lang.Math, Wrapper classes and associated methods (Number, Double, Float; Integer, Byte; Short, Long; Character, Boolean); Using java.util package: Core classes (Vector, Stack, Dictionary, Hashtable, Enumerations, Random Number Generation).

Unit 10: Holding Collection of Data [3 Hrs.]
Arrays And Collection Classes/Interfaces, Map/List/Set Implementations: Map Interface. 'List Interface, Set Interface, Collection Classes: Array List. Linked List, Hash Set and Tree Set; Accessing Collections/Use of An Iterator. Comparator.

Unit 11: Java Applications [8 Hrs.]
About AWT & Swing. About ,TFrame (top level window in Swing.). Swing components (JLabel, About text component like JTextField, JBttott, Event Handling in Swing Applications. Layout Management using Flow Layout.1Border Layout, Grid Layout, Using JPRel**.** Choice components 'ke .TCheck Box; JRadio Button, Borders components, JCombo Box & its events, JList & its events with MVC patterns, Key & Mouse Event Handling, Menus in swing, JText Area, Dialog boxes in swing, iTable for Displaying Data in Tabular form, MDI using JDesktop Pane & Enternal Frame, Using IDE like Netbeans, JBuildcr for building java applications using Drag & Drop), Adapter classes

Unit 12: Introduction to Java Applets [1 Hr.]
Definition, Applet lifecycle methods, Build a simple applet, Using Applet Viewer, Adding Controls: Animation Concepts.

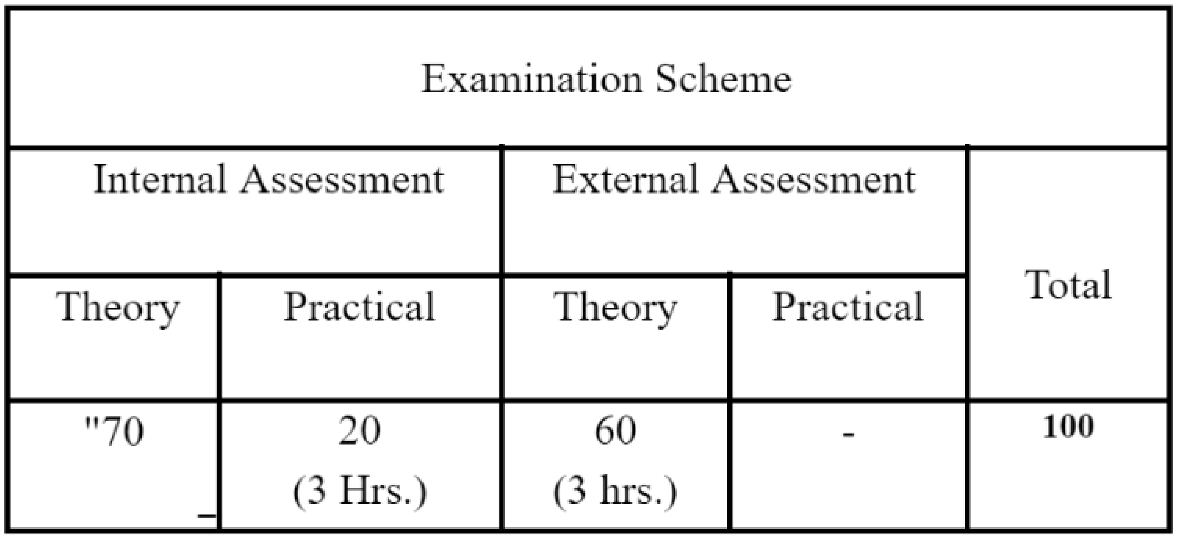
Unit 13: Database Programming using JDBC [2 Hrs.]
Using Connection, Statement & Result Set Interfaces for Manipulating Data with the Databases

Laboratory Works
Laboratory works should be done covering all the topics listed above and a small project work should be carried out using the concept learnt in this course. Project should be assigned on Individual Basis.

Teaching Methods
The general teaching pedagogy includes class lectures, group discussions, case studies, guest lectures, research work, project work, assignments (theoretical and practical), and examinations (written and

verbal), depending upon the nature of the topics. The teaching faculty will determine the choice of teaching pedagogy as per the need of the topics.

Evaluation



Examination Scheme				
Internal Assessment		External Assessment		Total
Theory	Practical	Theory	Practical	
"70	20 (3 Hrs.)	60 (3 hrs.)	-	**100**

Text Books
Deitel & Dietel. *"Java: How to program".* 9th Edition. Pearson Education. 2011, ISBN: 9780273754768 – McGraw-Hill 2006. ISn:; 0072263857

Reference Books
– Bruce Eckel, *"Thinking in Java",* 4 Edition, Prentice Hall, 2006, ISBN: 013-187248-6
– Cay Horstmann and Grazy Cornell, *"Core Java Volume 1-Fundamentals",* Ninth Edition, Prentice Hall, 2012, ISBN: 978-0137081899
– E. Balagurusamy, *"Programming with Java: A Primer",* Edition, Tata McGraw Hill Publication, India,