Java Programming

Comprehensive Notes on Java

By Virendra Goura

NOTE

Although every effort has been made to avoid errors and omissions, there is still a possibility that some mistakes may be missed due to invisibility.

This E - book is issued with the Understanding that the author is not responsible in any way for any errors/omissions.

BCA 301: Java Programming

Question Paper pattern for Main University Examination **Max Marks: 100**

Part-1 (very short answer) consists 10 questions of two marks each with two questions from each unit. Maximum limit for each question is up to 40 words.

Part-II (short answer) consists 5 questions of four marks each with one question from each unit. Maximum limit for each question is up to 80 words.

Part-III (Long answer) consists 5 questions of twelve marks each with one question from each unit with internal choice.

UNIT-I

Java Programming: Basic concept of object oriented programming (objects and classes, data abstraction & Encapsulation, Inheritance, polymorphism, dynamic binding, Message passing), Java features, JVM,Byte code interpretation, simple java program, Command line argument, Data types, type casting, operators (Arithmetic, increment, decrement, relational, Logical, bit wise, Conditional) and expressions.

UNIT- II

Decision Making and Branching: Decision making and branching (if....else, else if, switch). Looping, classes, objects and methods, constructor, wrapper classes, nesting of methods, overriding methods, final class, visibility control, arrays, strings.

UNIT-III

Inheritance & Multithreaded programming: Inheritance, types of Inheritance, Abstract class, interfaces, packages, multithreaded programming, extending thread, life cycle of thread, using thread methods, thread priority, synchronization.

UNIT-IV

Exception handling: Exception-handling fundamentals, Exception type, try, catch, throw, finally, creating exception sub classes.

AWT controls (Button, labels, Combo box, list and other Listeners), Layout and component managers, Event handling, string handling (Only main functions), Graphics programming (Line, rectangles, circle and ellipses).

UNIT-V

Overview of Networking in Java: URL class and its usage through connection, Sockets based connectivity, TCP/IP Sockets and Server sockets, Datagram Sockets.

Introduction to Java beans: BDK, JAR Files, Servlets Life Cycle of Servlet, JDBC connectivity.

UNIT-I Java Programming

Java Programming: Basic concept of object oriented programming (objects and classes, data abstraction & Encapsulation, Inheritance, polymorphism, dynamic binding, Message passing), Java features, JVM,Byte code interpretation, simple java program, Command line argument, Data types, type casting, operators (Arithmetic, increment, decrement, relational, Logical, bit wise, Conditional) and expressions.

---

1. Basic Concepts of Object-Oriented Programming (OOP)

Java is an object-oriented programming language, which means it is based on the concepts of **objects** and **classes**. Let's break down the key concepts of OOP:

- **Objects and Classes**:
  - **Class**: A class is a blueprint or template that defines the properties (attributes) and behaviors (methods) of an object. It defines how objects of that class will behave and what data they will store.

    ```
    class Car {
       String color;
       String model;
       void start() {
          System.out.println("Car is starting...");
       }
    }
    ```

  - **Object**: An object is an instance of a class. It is created based on the class definition and has actual values assigned to its attributes.

    ```
    Car myCar = new Car();
    myCar.color = "Red";
    myCar.model = "Sedan";
    myCar.start();  // Calls the start method of the Car class
    ```

- **Data Abstraction and Encapsulation**:
  - **Data Abstraction**: It is the concept of hiding the complex implementation details from the user and exposing only essential features of an object. It helps in focusing on the relevant details.
  - **Encapsulation**: It is the bundling of data (variables) and methods (functions) into a single unit, or class. It restricts direct access to some of an object's components and can prevent the accidental modification of data. This is achieved through access modifiers (private, public, protected).
  - class BankAccount {

    ```
    private double balance;  // private variable, cannot be accessed directly

    public void deposit(double amount) {  // public method
    ```

```
        if (amount > 0) {
            balance += amount;
        }
    }
```

- ```
    public double getBalance() {  // public method to access the balance
        return balance;
    }
}
```

- **Inheritance**:
  - Inheritance is a mechanism where one class can inherit properties and methods from another class. This allows for reusability and establishing a relationship between the parent class (superclass) and child class (subclass).

    ```
    class Animal {
        void eat() {
            System.out.println("Eating...");
        }
    }
    ```

  - ```
    class Dog extends Animal {
        void bark() {
            System.out.println("Barking...");
        }
    }
    ```

    - In this case, Dog inherits the eat() method from Animal.
- **Polymorphism**:
  - Polymorphism allows a single entity (method or object) to take multiple forms. It can be achieved in two ways:
    - **Method Overloading** (Compile-time Polymorphism): Multiple methods with the same name but different parameter lists.

      ```
      class Calculator {
          int add(int a, int b) {
              return a + b;
          }
      ```

```
    double add(double a, double b) {
        return a + b;
    }
}
```

- **Method Overriding** (Runtime Polymorphism): A subclass provides a specific implementation of a method that is already defined in its superclass.

```
class Animal {
    void sound() {
        System.out.println("Some sound...");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Barking...");
    }
}
```

- **Dynamic Binding**:
  - Dynamic binding refers to the process of linking a method call to its definition at runtime rather than compile time. It is used for method overriding and is essential for polymorphism.
- **Message Passing**:
  - In OOP, message passing refers to the communication between objects. It happens when an object sends a message to another object in the form of a method call.

---

2. Java Features

Java has several key features that make it a preferred programming language:

- **Platform Independence**: Java code is compiled into bytecode, which is executed by the Java Virtual Machine (JVM) on any platform, making it platform-independent.
- **Object-Oriented**: As discussed, Java is fully object-oriented, which helps in better code organization and maintenance.
- **Simplicity**: Java has a simple and easy-to-learn syntax. It eliminates complex features like pointers, operator overloading, etc.
- **Secure**: Java provides a secure environment through features like the absence of pointers and automatic memory management (garbage collection).

- **Multithreading**: Java supports multithreading, allowing the execution of multiple threads concurrently, which is important for performance and responsiveness.
- **Distributed Computing**: Java has built-in support for network applications and distributed computing (using Java RMI, sockets, etc.).

---

3. Java Virtual Machine (JVM) and Bytecode Interpretation

- **JVM (Java Virtual Machine)**:
  - The JVM is responsible for executing Java bytecode. It converts bytecode into machine code, allowing Java programs to run on any platform with a JVM.
  - **JVM Architecture** includes:
    - **Classloader**: Loads the classes into memory.
    - **Bytecode Verifier**: Verifies the bytecode for correctness.
    - **Execution Engine**: Executes the bytecode.
- **Bytecode**:
  - Bytecode is an intermediate representation of a Java program, compiled from the source code. It is independent of the hardware and operating system, making Java a platform-independent language.

---

4. Simple Java Program

Here's a simple example of a Java program:

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

- public class HelloWorld: Defines a class named HelloWorld.
- public static void main(String[] args): This is the entry point of the program. The main method is executed when the program is run.

---

5. Command-Line Arguments

Command-line arguments are values passed to a Java program at the time of execution.

Example:

```
public class CommandLine {
    public static void main(String[] args) {
        System.out.println("Arguments passed:");
```

```
        for (String arg : args) {
            System.out.println(arg);
        }
    }
}
```

If you run the program with arguments like this:

java CommandLine Hello World Java

Output:

Arguments passed:

Hello

World

Java

---

6. Data Types and Type Casting

- **Primitive Data Types**:
    - Java provides eight primitive data types: byte, short, int, long, float, double, char, and boolean.
- **Reference Data Types**:
    - These refer to objects and arrays, which are instances of classes.
- **Type Casting**:
    - Type casting refers to converting one data type into another. There are two types:
        - **Implicit Casting** (Widening): Automatically done by the compiler when converting from a smaller type to a larger type.

          int num = 10;

          double d = num; // Implicit casting
        - **Explicit Casting** (Narrowing): Manually converting from a larger type to a smaller type.

          double d = 10.5;

          int num = (int) d; // Explicit casting

---

7. Operators and Expressions

Java uses several types of operators to perform operations:

- **Arithmetic Operators**: Used to perform basic arithmetic operations.

  +, -, *, /, % (modulus)
- **Increment/Decrement Operators**: Increase or decrease a value by 1.

++, --

- **Relational Operators**: Used to compare two values.

  ==, !=, >, <, >=, <=

- **Logical Operators**: Used to perform logical operations.

  && (AND), || (OR), ! (NOT)

- **Bitwise Operators**: Operate on individual bits.

  &, |, ^, ~, <<, >>

- **Conditional (Ternary) Operator**: A shortcut for if-else statements.

  condition ? expr1 : expr2

Example:

```
int x = 10;

int y = 5;

int result = (x > y) ? x : y;  // If x > y, result = x; otherwise, result = y
```

- **Expressions**:
  - An expression is a combination of variables, operators, and methods that evaluates to a single value.

    Example: int result = (a + b) * c;

Conclusion

In this unit **Java Programming**, we covered the key concepts of Java, focusing on **object-oriented programming** (OOP) principles like **classes**, **objects**, **inheritance**, **polymorphism**, and **encapsulation**. These OOP features help in creating modular, reusable, and maintainable code. We also explored **Java features** such as platform independence through the **JVM**, and learned how **bytecode** is interpreted across different systems.

Additionally, we discussed **data types**, **type casting**, and **operators**, which are fundamental for manipulating data in Java. With these foundational concepts, you are equipped to write efficient Java programs and build scalable applications.

UNIT-II: Decision Making and Branching

Decision Making and Branching: Decision making and branching (if....else, else if, switch). Looping, classes, objects and methods, constructor, wrapper classes, nesting of methods, overriding methods, final class, visibility control, arrays, strings.

---

1. Decision Making and Branching

Java provides various control flow statements that allow you to make decisions and control the flow of execution based on conditions. The main decision-making statements in Java are if, else, else if, and switch.

if...else Statement: The if statement is used to execute a block of code if the condition is true; otherwise, it executes the code inside the else block.

```
if (x > 10) {
    System.out.println("x is greater than 10");
} else {
    System.out.println("x is less than or equal to 10");
}
```

- **else if Statement**: The else if statement is used when you have multiple conditions to check.

```
if (x > 10) {
    System.out.println("x is greater than 10");
} else if (x == 10) {
    System.out.println("x is equal to 10");
} else {
    System.out.println("x is less than 10");
}
```

- **switch Statement**: The switch statement is used when you need to compare a variable to multiple values. It is more efficient than using multiple if...else if statements when checking for many possible conditions.

```
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
```

```
        default:
            System.out.println("Invalid day");
    }
```

---

2. Looping

Java provides several looping structures that allow you to repeat a block of code multiple times.

- **for Loop**: The for loop is used when you know beforehand how many times you want the loop to run.

  ```
  for (int i = 0; i < 5; i++) {
      System.out.println(i);
  }
  ```

- **while Loop**: The while loop is used when you want the loop to continue as long as a condition is true.

  ```
  int i = 0;
  while (i < 5) {
      System.out.println(i);
      i++;
  }
  ```

- **do...while Loop**: The do...while loop is similar to the while loop, but it ensures that the loop will execute at least once.

  ```
  int i = 0;
  do {
      System.out.println(i);
      i++;
  } while (i < 5);
  ```

---

3. Classes, Objects, and Methods

- **Classes and Objects**: In Java, a class is a blueprint for creating objects (instances). An object represents a specific instance of a class with actual values for the attributes.

  ```
  class Car {
      String color;
      String model;

      void start() {
          System.out.println("Car is starting");
      }
  ```

```
}
```

- **Methods**: Methods define the behavior of an object. You can define methods within a class to perform actions.

```
class Car {
    String color;
    String model;

    void start() {
        System.out.println("Car is starting");
    }
}
```

---

4. Constructor

A **constructor** is a special method used to initialize objects. It is called when an object of a class is created.

- **Default Constructor**: If no constructor is defined, Java provides a default constructor.
- **Parameterized Constructor**: A constructor that takes arguments to initialize an object with specific values.

```
class Car {
    String color;
    String model;

    // Parameterized constructor
    Car(String c, String m) {
        color = c;
        model = m;
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Red", "Sedan");
    }
}
```

---

5. Wrapper Classes

Wrapper classes are used to wrap primitive data types into objects. Each primitive data type has a corresponding wrapper class.

- **Examples**:
    - int -> Integer
    - char -> Character
    - double -> Double
    - boolean -> Boolean

Wrapper classes allow primitive types to be used in situations that require objects, such as in collections or generics.

```
int x = 10;

Integer xObj = Integer.valueOf(x);  // Converting int to Integer
```

---

6. Nesting of Methods

In Java, it is possible to call one method from within another method. This is called **nesting of methods**.

```
class MathOperations {
    void add() {
        int sum = 10 + 20;
        System.out.println("Sum: " + sum);
    }

    void multiply() {
        add();  // Calling the add() method from multiply()
        int product = 10 * 20;
        System.out.println("Product: " + product);
    }
}
public class Main {
    public static void main(String[] args) {
        MathOperations mathOps = new MathOperations();
        mathOps.multiply();  // Calls multiply() which in turn calls add()
    }
}
```

---

7. Overriding Methods

Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This is used to achieve runtime polymorphism.

```
class Animal {
    void sound() {
        System.out.println("Some generic sound");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();
        animal.sound();  // Output will be "Barking"
    }
}
```

---

8. Final Class

The final keyword in Java is used to restrict the modification of classes, methods, and variables.

- **Final Class**: A final class cannot be extended (inherited).

  ```
  final class Car {
      // class definition
  }

  // This will cause a compile-time error:
  // class SportsCar extends Car {}
  ```

- **Final Method**: A final method cannot be overridden in subclasses.

- **Final Variable**: A final variable is a constant and its value cannot be changed once assigned.

---

9. Visibility Control

Java provides access modifiers to control the visibility and access level of classes, methods, and variables:

- **public**: Accessible from anywhere.
- **private**: Accessible only within the same class.
- **protected**: Accessible within the same package or subclasses.
- **default** (no modifier): Accessible within the same package.

Example:

```
class Car {
    public String model;  // Can be accessed anywhere
    private String engine;  // Can only be accessed within the Car class
}
```

---

10. Arrays

An **array** is a collection of similar data types stored in contiguous memory locations. In Java, arrays are objects, and they can store primitive data types or references to objects.

- **Declaring and Initializing Arrays**:

  ```
  int[] numbers = new int[5];  // Declares an array of integers with 5 elements
  numbers[0] = 10;  // Assigning value to the first element

  int[] numbers2 = {10, 20, 30, 40};  // Array initialized with values
  ```

- **Accessing Array Elements**:

  ```
  System.out.println(numbers2[0]);  // Access the first element (10)
  ```

---

11. Strings

In Java, strings are objects that represent a sequence of characters. Java provides a String class with various methods to manipulate strings.

- **String Declaration**:
- String message = "Hello, World!";
- **String Methods**:
  - length(): Returns the length of the string.
  - charAt(): Returns the character at a specified index.
  - substring(): Returns a substring from the original string.

```
String str = "Hello";

System.out.println(str.length());  // Output: 5

System.out.println(str.charAt(1));  // Output: e

System.out.println(str.substring(1, 4));  // Output: ell
```

Conclusion

In this unit, we explored important Java concepts such as decision making and branching (using if...else, else if, and switch), different types of loops (for, while, do...while), and essential object-oriented programming concepts including **classes**, **objects**, **constructors**, and **methods**. We also covered more advanced topics like **wrapper classes**, **method overriding**, **final classes**, and **visibility control**. Additionally, the unit introduced fundamental data structures such as **arrays** and **strings**. Mastering these concepts is crucial for building robust, efficient Java programs.

UNIT-III: Inheritance & Multithreaded Programming

Inheritance & Multithreaded programming: Inheritance, types of Inheritance, Abstract class, interfaces, packages, multithreaded programming, extending thread, life cycle of thread, using thread methods, thread priority, synchronization.

---

1. Inheritance in Java

Inheritance is one of the fundamental principles of Object-Oriented Programming (OOP). It allows a new class (subclass or child class) to inherit properties and behaviors (methods) from an existing class (superclass or parent class). This promotes reusability and establishes a relationship between classes.

Basic Inheritance: A subclass can inherit all the non-private members (fields and methods) from a superclass.

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}


class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}


public class Test {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // Inherited method from Animal class
        dog.bark(); // Method of Dog class
    }
}
```

2. Types of Inheritance

- **Single Inheritance**: When a class inherits from one superclass.

```
class Animal {
    void sound() {
        System.out.println("Some sound...");
```

```
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}
```

- **Multilevel Inheritance**: A class inherits from another class, which in turn inherits from another class.

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}

class Puppy extends Dog {
    void play() {
        System.out.println("Puppy is playing...");
    }
}
```

- **Hierarchical Inheritance**: When multiple classes inherit from the same superclass.

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}
```

```
class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("Meowing...");
    }
}
```

- **Multiple Inheritance** (not allowed in Java directly): In Java, a class cannot inherit from more than one class, to avoid ambiguity. However, Java supports multiple inheritance through interfaces, which we'll discuss later.

---

3. Abstract Class

An **abstract class** is a class that cannot be instantiated directly. It may contain abstract methods (methods without a body) as well as concrete methods (methods with a body).

Abstract Class Example:

```
abstract class Animal {
    abstract void sound(); // Abstract method (does not have a body)

    void eat() {          // Concrete method (has a body)
        System.out.println("Eating...");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Barking...");
    }
}

public class Test {
```

```
    public static void main(String[] args) {
        Animal dog = new Dog();
        dog.sound();  // Output: Barking...
        dog.eat();    // Output: Eating...
    }
}
```

---

4. Interfaces

An **interface** is a contract that a class must follow. It defines methods that must be implemented by the class that implements the interface. Unlike abstract classes, interfaces cannot have method implementations (except for default methods introduced in Java 8).

Interface Example:

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

class Dog implements Animal {
    public void sound() {
        System.out.println("Barking...");
    }

    public void eat() {
        System.out.println("Eating...");
    }
}

public class Test {
    public static void main(String[] args) {
        Animal animal = new Dog();
        animal.sound();  // Output: Barking...
        animal.eat();    // Output: Eating...
    }
}
```

- **Key Points**:
    - A class can implement multiple interfaces, providing a way to achieve multiple inheritance.
    - Interfaces allow for decoupling and flexibility.

---

5. Packages in Java

A **package** is a group of related classes, interfaces, and sub-packages. It helps in organizing code and avoiding name conflicts. There are two types of packages in Java:

- **Built-in Packages**: Java provides many built-in packages like java.util, java.io, etc.
- **User-defined Packages**: Packages created by developers to organize classes.

Creating a Package:

```
package com.example;


public class Car {
    public void start() {
        System.out.println("Car is starting...");
    }
}
```

Importing a Package:

```
import com.example.Car;


public class Test {
    public static void main(String[] args) {
        Car car = new Car();
        car.start();
    }
}
```

---

6. Multithreaded Programming

Multithreading is the concurrent execution of more than one part of a program to maximize the utilization of CPU. Java provides built-in support for multithreading, allowing you to run multiple threads concurrently.

- **Thread Creation**: There are two ways to create a thread:
    1. **By extending the Thread class**.
    2. **By implementing the Runnable interface**.

Extending the Thread Class:

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

public class Test {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();  // Start the thread
    }
}
```

Implementing the Runnable Interface:

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

public class Test {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        thread.start();
    }
}
```

---

7. Life Cycle of a Thread

A thread in Java goes through several states during its life cycle:

1. **New**: When a thread is created but not yet started.
2. **Runnable**: When the thread is ready to run and waiting for CPU time.
3. **Blocked**: When the thread is blocked due to I/O or waiting for resources.
4. **Waiting**: When a thread is waiting for another thread to perform a particular action.

5. **Terminated**: When a thread has finished its execution.

Thread Methods:

- start(): Begins the execution of the thread.
- run(): Contains the code that constitutes the thread's task.
- sleep(long milliseconds): Pauses the thread for the specified time.
- join(): Ensures that one thread finishes before another starts.

---

8. Thread Priority

Threads in Java have a priority, which determines the order in which threads are scheduled to run. The priority is set using the Thread class:

- **Priority Range**: 1 (lowest) to 10 (highest).
- **Default Priority**: The default priority is 5.

Setting Thread Priority:

```
Thread thread = new Thread();

thread.setPriority(Thread.MAX_PRIORITY);  // Set highest priority
```

Thread Scheduler: The operating system's thread scheduler manages thread priorities and determines the order of execution.

---

9. Synchronization

Synchronization is a technique to ensure that only one thread can access a resource at a time. This prevents issues like data inconsistency in a multithreaded environment.

Synchronized Method:

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

   synchronized void increment() {
      count++;
   }
}
```

Synchronized Block: A more granular form of synchronization, which allows you to synchronize only specific code blocks.

```
synchronized (object) {
   // critical section code
}
```

Why Use Synchronization?

- To prevent **race conditions** where two threads simultaneously modify shared data.
- To ensure **data consistency** when multiple threads access shared resources.

Conclusion

In this unit, we covered essential concepts of **Inheritance** and **Multithreaded Programming** in Java. Inheritance enables code reuse and the creation of relationships between classes. Java supports multiple types of inheritance, including single, multilevel, and hierarchical inheritance. We also learned about **abstract classes**, **interfaces**, and **packages**, all of which help organize and structure code efficiently.

In the realm of **multithreaded programming**, we explored how to create and manage threads using the Thread class and Runnable interface, the life cycle of a thread, setting thread priorities, and using synchronization to prevent data inconsistency. Mastering these concepts will help in building more efficient and concurrent applications in Java.

UNIT-IV: Exception Handling and AWT

Exception handling: Exception-handling fundamentals, Exception type, try, catch, throw, finally, creating exception sub classes.

AWT controls (Button, labels, Combo box, list and other Listeners), Layout and component managers, Event handling, string handling (Only main functions), Graphics programming (Line, rectangles, circle and ellipses).

---

1. Exception Handling in Java

Exception handling in Java is a mechanism to handle runtime errors, ensuring that the normal flow of the application is maintained. Java provides a set of constructs for handling exceptions, such as try, catch, throw, finally, and creating custom exception classes.

Exception Fundamentals

An exception is an event that disrupts the normal flow of the program. Java uses a try-catch block to handle exceptions.

```
try {
    // Code that might throw an exception
    int result = 10 / 0; // ArithmeticException
} catch (ArithmeticException e) {
    // Handling exception
    System.out.println("Error: " + e.getMessage());
}
```

Exception Types

1. **Checked Exceptions**: These exceptions are checked at compile time. They are subclasses of Exception, except for RuntimeException. For example, IOException, SQLException.
2. **Unchecked Exceptions**: These exceptions occur during runtime and are subclasses of RuntimeException. For example, ArithmeticException, NullPointerException.
3. **Errors**: These are abnormal conditions that cannot be handled by a program (e.g., OutOfMemoryError, StackOverflowError).

try, catch, and finally

- **try**: The block of code where exceptions might occur.

```
try {
    int result = 10 / 0;  // ArithmeticException
}
```

- **catch**: Catches exceptions thrown by the try block and provides handling.

```
catch (ArithmeticException e) {
    System.out.println("Error: " + e.getMessage());
}
```

- **finally**: The block that will execute regardless of whether an exception occurs or not. It is used to release resources like closing files or database connections.

```
finally {
    System.out.println("This will always execute.");
}
```

Throwing Exceptions

You can throw exceptions using the throw keyword. This is often used when custom error conditions need to be raised.

```
throw new ArithmeticException("Division by zero is not allowed");
```

Creating Exception Subclasses

You can create custom exceptions by subclassing the Exception class.

```
class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}

public class Test {
    public static void main(String[] args) throws CustomException {
        throw new CustomException("This is a custom exception");
    }
}
```

---

2. AWT Controls and Event Handling

Abstract Window Toolkit (AWT) is a set of APIs used for building graphical user interfaces in Java. AWT provides several controls like buttons, labels, text fields, combo boxes, and more, for building user interfaces.

AWT Controls

- **Button**: A button that can be clicked to perform actions.

```
Button button = new Button("Click Me");
button.setBounds(50, 50, 150, 30);
```

- **Label**: A simple text element for displaying information.

```
Label label = new Label("Hello, AWT!");
label.setBounds(50, 100, 100, 30);
```

- **Combo Box**: A drop-down list of options.

```
Choice choice = new Choice();
choice.add("Option 1");
choice.add("Option 2");
choice.setBounds(50, 150, 150, 30);
```

- **List**: A list with multiple options.

```
List list = new List();
list.add("Item 1");
list.add("Item 2");
list.setBounds(50, 200, 150, 60);
```

Event Handling

Event handling in Java is done by attaching listeners to components. These listeners capture user actions (like button clicks, mouse events, etc.) and respond to them.

For example, to handle a button click:

```
Button button = new Button("Click Me");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button Clicked!");
    }
});
```

- **MouseListener**: Captures mouse events (click, press, release).
- **KeyListener**: Captures keyboard events (key press, release).

Layout and Component Managers

AWT uses **layout managers** to manage the arrangement of components within a container. Some common layout managers are:

1. **FlowLayout**: Arranges components in a left-to-right flow.

```
Frame f = new Frame();
f.setLayout(new FlowLayout());
```

2. **BorderLayout**: Divides the container into five regions (North, South, East, West, Center).

```
Frame f = new Frame();
f.setLayout(new BorderLayout());
```

3. **GridLayout**: Arranges components in a grid of rows and columns.

```
Frame f = new Frame();
f.setLayout(new GridLayout(2, 2));
```

3. String Handling in Java

String handling in Java refers to operations that can be performed on **String objects**. Java provides a String class with methods to perform various string manipulations, such as concatenation, comparison, and searching.

Some of the commonly used **String functions**:

- length(): Returns the length of the string.

```
String str = "Hello";

int len = str.length(); // 5
```

- concat(): Concatenates two strings.

```
String str1 = "Hello";

String str2 = "World";

String result = str1.concat(str2); // "HelloWorld"
```

- substring(): Extracts a part of the string.

```
String str = "Hello World";

String sub = str.substring(0, 5); // "Hello"
```

- toUpperCase(): Converts the string to uppercase.

```
String str = "hello";

String upper = str.toUpperCase(); // "HELLO"
```

- toLowerCase(): Converts the string to lowercase.

```
String str = "HELLO";

String lower = str.toLowerCase(); // "hello"
```

- equals(): Compares two strings for equality.

```
String str1 = "hello";

String str2 = "hello";

boolean isEqual = str1.equals(str2); // true
```

---

4. Graphics Programming

Graphics programming in Java is done using the **Graphics class**, which is part of the AWT library. It provides methods to draw shapes and images on the screen.

Drawing Basic Shapes

- **Line**:

```
Graphics g = getGraphics();

g.drawLine(20, 30, 100, 200);
```

- **Rectangle**:

g.drawRect(50, 50, 200, 100); // Drawing rectangle at (50,50) with width 200 and height 100

- **Circle**:

g.drawOval(100, 100, 50, 50); // Drawing a circle with diameter 50

- **Ellipse**:

g.drawOval(100, 100, 200, 100); // Drawing an ellipse with width 200 and height 100

Graphics Example

Here is an example of a simple Java program to draw a line, rectangle, circle, and ellipse:

```
import java.awt.*;
import java.awt.event.*;


public class GraphicsExample extends Frame {
    public void paint(Graphics g) {
        g.drawLine(50, 50, 200, 50); // Drawing a line
        g.drawRect(50, 100, 150, 75); // Drawing a rectangle
        g.drawOval(50, 200, 100, 100); // Drawing a circle
        g.drawOval(200, 200, 150, 75); // Drawing an ellipse
    }

    public static void main(String[] args) {
        GraphicsExample ge = new GraphicsExample();
        ge.setSize(400, 400);
        ge.setVisible(true);
    }
}
```

Conclusion

In this unit, we covered essential concepts of **Exception Handling**, **AWT Controls**, and **Graphics Programming**. Java's exception handling mechanism allows you to handle errors gracefully using try, catch, throw, and finally. We also explored various **AWT controls** for building graphical user interfaces and handling user events, as well as **layout managers** to organize components. Lastly, we learned how to perform basic **graphics programming** to draw shapes like lines, rectangles, circles, and ellipses using the Graphics class. Mastering these concepts will help in building robust applications with graphical interfaces and proper error handling.

UNIT-V: Networking, Java Beans, Servlets, and JDBC

Overview of Networking in Java: URL class and its usage through connection, Sockets based connectivity, TCP/IP Sockets and Server sockets, Datagram Sockets.

Introduction to Java beans: BDK, JAR Files, Servlets Life Cycle of Servlet, JDBC connectivity.

---

1. Overview of Networking in Java

Networking in Java allows applications to communicate with each other over a network. Java provides a robust API to handle networking tasks.

URL Class and Its Usage

The URL class is used to represent a Uniform Resource Locator (URL) and establish a connection to the resource pointed to by the URL.

Example of using the URL class to open a connection:

```
import java.net.*;

import java.io.*;


public class URLExample {

    public static void main(String[] args) throws Exception {

        URL url = new URL("http://www.example.com");

        BufferedReader reader = new BufferedReader(new InputStreamReader(url.openStream()));


        String line;

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

            System.out.println(line);

        }

        reader.close();

    }

}
```

Sockets Based Connectivity

Java provides two types of socket communication:

- **TCP/IP Sockets**: Reliable, connection-oriented communication using streams.
- **Datagram Sockets**: Used for connectionless communication (UDP).

TCP/IP Sockets

TCP/IP sockets provide a two-way communication channel between client and server. You can use Socket for the client-side and ServerSocket for the server-side.

Server Side:

```
import java.net.*;
import java.io.*;


public class ServerExample {
    public static void main(String[] args) throws Exception {
        ServerSocket server = new ServerSocket(1234);
        Socket client = server.accept(); // Accept client connection
        BufferedReader reader = new BufferedReader(new InputStreamReader(client.getInputStream()));
        String message = reader.readLine();
        System.out.println("Message from client: " + message);
    }
}
```

Client Side:

```
import java.net.*;
import java.io.*;


public class ClientExample {
    public static void main(String[] args) throws Exception {
        Socket socket = new Socket("localhost", 1234);
        PrintWriter writer = new PrintWriter(socket.getOutputStream(), true);
        writer.println("Hello, Server!");
    }
}
```

Datagram Sockets

Datagram sockets are used for sending and receiving data without establishing a connection (UDP). It's faster but less reliable than TCP.

Example of Datagram Socket:

```
import java.net.*;


public class DatagramExample {
    public static void main(String[] args) throws Exception {
        DatagramSocket socket = new DatagramSocket();
        String message = "Hello, Datagram!";
```

```
        DatagramPacket packet = new DatagramPacket(message.getBytes(), message.length(), InetAddress.getByName("localhost"), 1234);
        socket.send(packet);
    }
}
```

---

2. Introduction to Java Beans

JavaBeans are reusable software components that can be manipulated in visual development environments. They follow specific conventions to allow them to be easily used by other components.

- **BDK (Bean Development Kit)**: A tool that allows developers to create, debug, and test JavaBeans.
- **JAR (Java Archive) Files**: A JAR file is a compressed package that can contain classes, images, and other resources. JavaBeans are often packaged as JAR files for distribution.

jar cf bean.jar MyBean.class

---

3. Servlets and Servlet Life Cycle

Servlets are server-side Java programs that handle client requests and generate dynamic responses. They run inside a servlet container (like Tomcat or Jetty).

Life Cycle of a Servlet

1. **Initialization**: The servlet is initialized once when the server starts. The init() method is called.
2. **Request Handling**: The servlet processes client requests in the service() method.
3. **Destruction**: The servlet is destroyed when the server shuts down, calling the destroy() method.

Example Servlet:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorldServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<h1>Hello, World!</h1>");
    }
}
```

Servlet Life Cycle Methods

- init(): Called once when the servlet is loaded into memory.
- service(): Handles the client request and generates a response.
- destroy(): Cleans up resources when the servlet is about to be removed.

---

4. JDBC Connectivity

JDBC (Java Database Connectivity) allows Java applications to interact with relational databases. It provides an API for connecting to databases and executing SQL queries.

Steps for JDBC Connectivity

1. **Load the Driver**: Load the database driver class.
2. **Establish a Connection**: Use the DriverManager to establish a connection to the database.
3. **Create a Statement**: Create a Statement object to send SQL queries to the database.
4. **Execute the Query**: Execute SQL queries using the executeQuery() or executeUpdate() methods.
5. **Process the Results**: Retrieve and process the result set.
6. **Close the Connection**: Always close the connection to release database resources.

Example of JDBC connectivity:

```
import java.sql.*;


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

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

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

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

            // Process the results
```

```
            while (rs.next()) {
                System.out.println("User ID: " + rs.getInt("id"));
                System.out.println("User Name: " + rs.getString("name"));
            }

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

Conclusion

In this unit, we covered key concepts related to **Networking**, **JavaBeans**, **Servlets**, and **JDBC**. Java networking allows for client-server communication using sockets and URLs. We also explored **JavaBeans**, which are reusable components, and how they can be packaged in **JAR** files. The **Servlet life cycle** gives an overview of how servlets handle client requests. Finally, **JDBC** enables database connectivity in Java, allowing you to perform database operations using SQL queries. Mastering these topics will help you build networked, database-driven, and reusable Java applications.