DrillBit

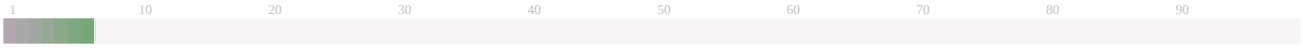
The Report is Generated by DrillBit Plagiarism Detection Software

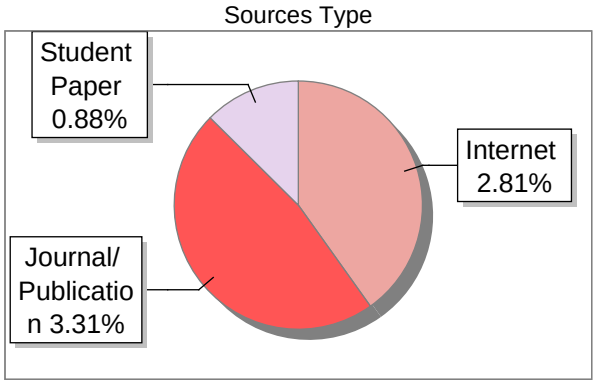
Submission Information

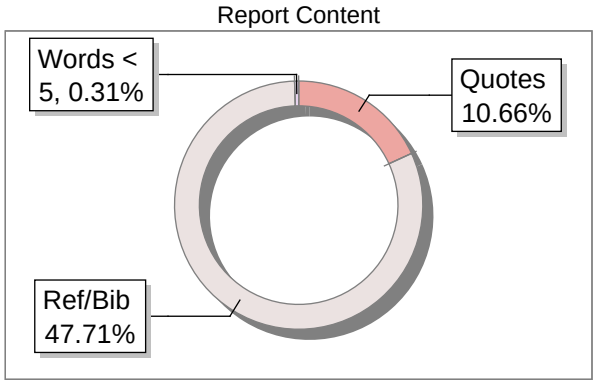
Author Name	Bimal Kumar Kalita
Title	JAVA
Paper/Submission ID	2996678
Submitted by	librarian.adbu@gmail.com
Submission Date	2025-01-20 12:13:20
Total Pages, Total Words	175, 33796
Document type	Others

Result Information

Similarity **7 %**







Exclude Information

Quotes	Excluded
References/Bibliography	Excluded
Source: Excluded < 5 Words	Excluded
Excluded Source	**0 %**
Excluded Phrases	Not Excluded

Database Selection

Language	English
Student Papers	Yes
Journals & publishers	Yes
Internet or Web	Yes
Institution Repository	Yes

A Unique QR Code use to View/Download/Share Pdf File

DrillBit

DrillBit Similarity Report

7	47	A
SIMILARITY %	MATCHED SOURCES	GRADE

A-Satisfactory (0-10%)
B-Upgrade (11-40%)
C-Poor (41-60%)
D-Unacceptable (61-100%)

LOCATION	MATCHED DOMAIN	%	SOURCE TYPE
1	www.dbuniversity.ac.in	2	Publication
2	www.dbuniversity.ac.in	1	Publication
3	index-of.es	<1	Publication
5	pdfcookie.com	<1	Internet Data
6	pdfcookie.com	<1	Internet Data
8	pdfcookie.com	<1	Internet Data
9	mu.ac.in	<1	Publication
10	pdfcookie.com	<1	Internet Data
13	Submitted to U-Next Learning on 2024-07-15 18-32 2114892	<1	Student Paper
14	stackify.com	<1	Internet Data
15	pdfcookie.com	<1	Internet Data
16	byjus.com	<1	Publication
19	Submitted to U-Next Learning on 2024-07-07 20-22 2087984	<1	Student Paper
20	Submitted to U-Next Learning on 2025-01-07 20-35 2951726	<1	Student Paper

21	raygun.com	<1	Internet Data
22	www.prepbytes.com	<1	Internet Data
23	Submitted to U-Next Learning on 2024-07-12 18-49 2108002	<1	Student Paper
26	Submitted to U-Next Learning on 2024-11-25 15-51 2582029	<1	Student Paper
27	technodocbox.com	<1	Internet Data
28	qdoc.tips	<1	Internet Data
30	www.upgrad.com	<1	Internet Data
31	index-of.es	<1	Publication
34	www.rose-hulman.edu	<1	Publication
35	moam.info	<1	Internet Data
36	moam.info	<1	Internet Data
40	Submitted to U-Next Learning on 2024-07-02 10-54 2072073	<1	Student Paper
41	stackify.com	<1	Internet Data
42	index-of.es	<1	Publication
45	index-of.es	<1	Publication
46	ncert.nic.in	<1	Publication
47	moam.info	<1	Internet Data
51	gnit.ac.in	<1	Internet Data
52	pdfcookie.com	<1	Internet Data

56	fdokumen.id	<1	Internet Data
58	blogs.umass.edu	<1	Publication
59	moam.info	<1	Internet Data
61	moam.info	<1	Internet Data
62	pdfcookie.com	<1	Internet Data
65	qdoc.tips	<1	Internet Data
67	www.rajagiritech.ac.in	<1	Publication
70	Educating for critical thinking thoughtencouraging questions in a community of by Golding-2011	<1	Publication
72	pdfcookie.com	<1	Internet Data
73	pdfcookie.com	<1	Internet Data
74	pgc.mahajana.edu.in	<1	Publication
75	Submitted to U-Next Learning on 2024-12-03 16-53 2673963	<1	Student Paper
76	thenewstack.io	<1	Internet Data
77	tnou.ac.in	<1	Publication

Master of Computer Application (Online)

Subject: CAOPJ0018: PROGRAMMING THROUGH JAVA

(4 credits–120 hours)

Online Course Material- SLM (Student Learning Material)

Prepared by- Course Co-ordinator

Dr. Bimal Kumar Kalita
Assistant Professor (Selection)
Assam Don Bosco University

PROGRAMMING THROUGH JAVA

Objective

The course is designed to impart the knowledge and skill required to solve real world problems using an object-oriented approach utilizing Java language constructs. This course covers the two main parts of Java i.e. Java Language and Java Library (JDK 5). After completion of the course, a student is expected to be able to

- Do Object Oriented Programming using Java
- Implement Exception handling and Multithreading in Java.
- Create Java I/O Applications and Applets. Set up a GUI using Swing components Do Network Programming in Java.
- Access relational databases from the Java program and use Java Beans and Servlets.

COURSE / LEARNING OUTCOMES

At the end of this course students will be able to:

1. Recall the various features of Object Oriented programming by utilizing the JAVA language construct. (Remembering) 2. Explain the standard library, scope and lifetime of a variable and various control statements used in JAVA programs. (Understanding)
3. Interpret the concept of classes and object in JAVA and apply exception handling to solve various exceptions (Applying)
4. Contrast the different type of inheritance and polymorphism and Analyse it in resolving various problems (Analysing)
5. Select the appropriate GUI and will be able to justify their decision to use a particular GUI by evaluating the required parameters depending on the domain and requirement. (Evaluating)
6. Develop algorithms based on the knowledge they have gained to design cost effective and user friendly applications. (Creating)

Module I: Core Java Programming (14 Hours)

Unit 1:

Java Overview: Genesis, Java Philosophy, Java and Internet, Object-Oriented Programming features, Java Applet and Application, Java Environment and Java Development Kit (JDK) and Java Standard Library (JSL).

Unit 2:

Java language fundamentals: Lexical units, constants & variables, The scope and lifetime of variable, Data Types, Type conversion: Coercion and casting, Control statements, Arrays.

Unit 3:

Classes and objects: The this keyword, Garbage collection, Overloading constructor, Using object as parameters, Argument passing, Returning objects, Recursion, Introducing Access control (public, private and protected), static, final, nested classes, String class, Command-line argument

Module II: Inheritance, Exception handling, Multithread and Applets (12 Hours)

Unit 4:

Inheritance: Member access and inheritance, method overriding, dynamic method dispatch, using abstract classes, using final with inheritance, the Object class; Packages, Interface, classpath.

Unit 5:

Exception handling: Fundamentals, Exception types, Java's built-in exceptions, user defined exceptions.

Unit 6:

Multithreaded Programming: The Java thread model (thread priorities, synchronization and inter-thread communication); Deadlock, Thread Group.

Unit 7:

I/O Basics: Streams, the stream classes, the predefined streams, Reading console input, writing console output, the transient and volatile modifiers, using instance of native methods.

Module III: String handling, Utility classes, java.lang and java.io (12 Hours)

Unit 8:

String handling: String constructors, methods for character extraction, string searching and comparison, data conversion using valueof (), String Buffer.

Unit 9:

Exploring java. lang: Simple type wrappers, System class, class Class, Math functions

Unit 10:

The utility classes: Vector, Stack, Hash Table, String Tokenizer, Bit set, Date, Calendar, Gregorian Calendar,
Random, Observable

Unit 11:

Input/Output: Exploring java.io: The java.io classes and interface, File class and methods for creating, renaming, listing and deleting files and directories, I/O stream classes (File Input Sream, File Output Stream, Buffered Input Stream, Buffered Output Stream, Push Back Input Stream, Input Stream Reader, Buffered Reader, Buffered Writer, Print Stream, Random Access File)

Module IV: Networking, Images, Applet class and Swing (12 Hours)

Unit 12:

Networking: Socket overview, Stream Sockets, Datagram sockets, Manipulating URLs, Establishing a simple Server/Client using Stream Sockets, Connectionless Client/Server Interaction with Datagrams

***Unit 13*:**

Images: File formats, image fundamentals, creating, loading and displaying images, ImageObserver, MediaTracker

Unit 14:

The Applet class: applet architecture, passing parameters to applets, getDocumentBase, getCodeBase, and showDocument, AppletContext and AudioClip interfaces, Graphics class and methods for drawing lines, rectangles, polygons and ovals

14.1 Swing: Component and Container classes, Layout managers (Flow Layout, Grid Layout, Border Layout), Handling events, Adapter classes, Anonymous inner classes

14.2 Swing GUI components: JLabel, JTextField, JTextArea, JButton, JCheckBox, JRadioButton, JList, JComboBox, JScrollBar, JScrollPane, JToolTip, JPanel, JFrame

14.3 Menus: JmenuBar, JMenu, JMenuItem, JSeparator

Module V: Java Beans, JDBC, Java Servlets (10 Hours)

Unit 15:

Java Beans: Introducing JavaBeans Concepts and Bean Development Kit (BDK), Using the Bean Box, Writing a simple Bean,
Bean Properties (simple properties), Manipulating events in the Bean Box

Unit 16:

Java database connectivity (JDBC): Introduction to JDBC, type of JDBC connectivity, Establishing database connections, Accessing relational database from Java programs

DETAILS OF MODULE 1:

Module I: Core Java Programming (14 Hours)

Unit 1:

Java Overview: Genesis, Java Philosophy, Java and Internet, Object-Oriented Programming features, Java Applet and Application, Java Environment and Java Development Kit (JDK) and Java Standard Library (JSL).

Unit 2:

Java language fundamentals: The scope and lifetime of variable, Type conversion and casting, Control statements, Arrays.

Unit 3:

Classes and objects: The this keyword, Garbage collection, Overloading constructor, Using object as parameters, Argument passing, Returning objects, Recursion, Introducing Access control (public, private and protected), static, final, nested classes, String class, Command-line argument

Unit 1:

1. Java Overview:

1.1 Introduction
1.2 Unit overview
1.3 Java Philosophy, Java and Internet, & Object-Oriented Programming features,
1.4 Java Applet and Application
1.5 Java Environment and Java Development Kit (JDK)
1.6 Java Standard Library (JSL).
1.7 Unit Summary
1.8 Check your progress

1.1 **Introduction to Java:** It is a fascinating story about innovation and adaptation in the software industry. Java, one of the world's most popular programming languages, was developed in the early 1990s by a team led by James Gosling at Sun Microsystems. Here's a content outline covering its origins and evolution:

Early Beginnings and Motivation

- **Context in the Early 90s**: Personal computers were becoming popular, and the internet was on the rise. There was a demand for portable, cross-platform software solutions that could adapt across different systems.
- **Sun Microsystems' Goals**: Initially, Sun Microsystems aimed to create a language for interactive television and digital devices (like set-top boxes) but pivoted when they saw the broader potential.
- **Project Name – Green Project**: Java started as part of an internal research project called "Green Project," aiming to develop a language that could operate across various electronic devices.

Key People and Development

- **James Gosling and His Team**: Gosling, along with Mike Sheridan and Patrick Naughton, was instrumental in developing Java. They created the language while experimenting with a project called "Oak," named after an oak tree outside Gosling's office.
- **Switch from Oak to Java**: The language was renamed Java due to trademark issues with the name "Oak." The new name "Java" was inspired by coffee, as the team often drank Java coffee during development.

1.2 Overview:

1.3 Foundational Design Principles of Java, Java Philosophy:

Java's philosophy revolves around several core principles that guide its design and implementation:

1. **Write Once, Run Anywhere (WORA)**: Java aims to be platform-independent, meaning that code written in Java can run on any device with a Java Virtual Machine (JVM). This is enabled by compiling Java code into bytecode, which can be interpreted on any operating system.
2. **Object-Oriented**: Java is designed as a fully object-oriented programming language, encouraging modularity and reusability of code. Concepts like classes, inheritance, and polymorphism are central, making Java suitable for complex software development.
3. **Simplicity and Ease of Use**: Java's syntax is designed to be clear and readable, making it accessible for both beginners and experienced programmers. The language also manages many complex functions (like memory management) through features like automatic garbage collection.
4. **Robustness and Security**: Java includes strong memory management features and a security model designed to prevent common programming errors. Its architecture minimizes vulnerabilities and has built-in security measures, such as the sandbox model for applets, which prevent untrusted code from harming the system.
5. **Multithreaded and High Performance**: Java supports multithreading, allowing multiple threads to run concurrently, which is essential for modern applications. Its performance is enhanced by Just-In-Time (JIT) compilation and optimized libraries.
6. **Dynamic and Versatile**: Java is highly adaptable, supporting dynamic loading of classes, and evolving over time to incorporate new libraries, frameworks, and tools that support web, mobile, and enterprise applications.

Together, these principles make Java a popular choice for building robust, portable, and high-performance applications across a wide variety of platforms.

Java’s Rise to Popularity

- **Browser Integration with Applets**: When the internet gained traction, Java applets, which could run in web browsers, became a popular feature. This helped Java gain visibility and rapid adoption.
- **Enterprise and Backend Adoption**: Java’s stability and platform independence made it ideal for enterprise-level applications, leading to widespread adoption in backend development and server-side applications.

Java in the Modern Era

- **Open Source and Oracle’s Acquisition**: Sun Microsystems was acquired by Oracle in 2010, and Java became open source, which broadened its community and support.
- **Java’s Evolution**: Over the years, Java has adapted with features like lambda expressions, modularity (Java 9), and faster release cycles.

- **Use in Android and Beyond**: Java became a primary language for Android development, expanding its reach into mobile applications.

Java's Enduring Legacy and Future

- **Java Today**: Java remains a top choice for web development, enterprise applications, big data, IoT, and cloud computing.
- **Ongoing Developments**: With the latest versions and features, Java continues to be a vital programming language in both industry and academia.
- **Looking Forward**: Java's adaptability, community support, and frequent updates signal that it will continue to evolve and remain relevant.

Java and Internet:

Java is a versatile programming language that plays a significant role in web development and internet applications. The portability feature has made Java ideal for internet applications, especially in environments requiring high performance and scalability.

In the early days of the web, Java applets were popular for adding interactive features to web pages. Although applets are now mostly obsolete due to security concerns and better alternatives, Java remains crucial for server-side development. Java-based frameworks like Spring and JavaServer Pages (JSP) are widely used in building complex web applications and APIs, powering backend services and enterprise-level applications. Java's extensive libraries, security features, and platform independence continue to make it a top choice for developing robust internet applications, from websites to mobile apps to cloud services.

Java's Object Oriented features:

Java is a popular object-oriented programming (OOP) language that follows the principles of OOP, which are designed to increase code modularity, flexibility, and reusability. Here are its core OOP properties:

1. **Encapsulation**: This is the bundling of data (attributes) and methods (functions) that operate on that data within a single unit or class. Java uses access modifiers (like private, protected, and public) to control access, ensuring that only the class's methods can modify its internal state directly. Encapsulation helps protect the internal data structure and provides a clear interface for interaction.
2. **Inheritance**: Java allows classes to inherit attributes and behaviors (methods) from other classes, promoting code reuse and hierarchy structure. The extends keyword enables a class to inherit from a parent (or superclass), which reduces redundancy. For instance, a Car class could inherit general vehicle properties from a Vehicle class. This makes code more maintainable and organized.
3. **Polymorphism**: Java supports polymorphism, allowing objects to be treated as instances of their parent class or interface. This can be achieved through method overloading (same method name, different parameters) and method overriding (modifying a method's

behavior in a subclass). It enables flexibility and integration of various classes in a unified way, as objects can take on multiple forms.

4. **Abstraction**: Abstraction in Java is the concept of hiding complex implementation details and exposing only essential features. Java supports abstraction using abstract classes (abstract keyword) and interfaces, where interfaces provide complete abstraction by allowing classes to implement multiple behaviors while decoupling the details of those behaviors. This allows developers to focus on "what" a class does rather than "how" it does it.

These principles make Java a robust, maintainable, and extensible language well-suited for a variety of applications.

.1.5 **Java Applet**

Java applets and Java applications are two types of Java programs, each with distinct characteristics, purposes, and methods of execution

- **Definition**: A Java applet is a small Java program that runs within a web browser, typically embedded in HTML and loaded over the internet.
- **Execution**: Applets run inside a Java-enabled browser or a web viewer that supports applet execution, relying on a Java Virtual Machine (JVM) within the browser.
- **Security**: Due to security risks, applets operate in a restricted environment, known as the "sandbox," which limits their ability to interact with the system directly (e.g., file access, network connections).
- **Uses**: Applets were popular for adding interactivity to web pages (e.g., games, calculators) but are now obsolete due to security concerns and the evolution of web technologies like JavaScript, HTML5, and CSS.
- **Deployment**: Distributed via a webpage; the browser loads the applet using the <applet> tag or an object/embed tag in HTML.

Java Application

- **Definition**: A Java application is a standalone program that runs directly on the user's computer or server.
- **Execution**: It executes in a standard JVM without relying on a web browser. Applications are usually launched through the command line or a graphical interface.
- **Security**: Since they don't run in a restricted environment, Java applications have broader access to system resources, though they must still comply with JVM security policies.
- **Uses**: Applications are used for various purposes, from business software to games and utilities, and can be simple command-line programs or complex GUI applications.
- **Deployment**: Distributed as standalone .jar (Java Archive) files or bundled with an installer. They can be executed directly with java <AppName> from the command line or through a GUI.

Key Differences:

Feature	Java Applet	Java Application
Environment	Web browser (deprecated)	Directly on OS, standalone
Security	Sandbox, restricted	Full system access, within JVM restrictions
Usage	Web-based interactivity	Full-fledged software, desktop apps
Lifecycle	Controlled by the browser	Controlled by the application code
Deployment	Embedded in HTML, run in browsers	Distributed as .jar files or executables

Note: Due to security issues and modern web standards, Java applets are largely unsupported in today's browsers and are generally considered outdated.

1.6 Java Environment & Java Development Kit (JDK):

In Java, the **Java Environment** and the **Java Development Kit (JDK)** are essential parts of developing and running Java applications. Here's an overview of each:

Java Environment:

The Java environment includes everything necessary to write, compile, and execute Java programs. It is typically broken down into several main components:

- **Java Runtime Environment (JRE):** The JRE is part of the Java environment that is used to run Java applications. It includes:
    - **Java Virtual Machine (JVM):** The JVM is responsible for executing Java bytecode, making Java applications platform-independent. It provides an environment where Java code can run, with built-in memory management and garbage collection.
    - **Java Class Libraries:** Pre-built classes (such as java.lang, java.util, etc.) that provide essential functions like file I/O, networking, and basic data structures.

  Note: The JRE is enough to run Java programs but not to develop them from scratch, as it lacks development tools.

Java Development Kit (JDK)

The JDK is a complete package that includes everything in the JRE and additional tools for developing Java applications. Here are the main components of the JDK:

- **JRE Components:** Since the JDK includes the JRE, it has everything needed to execute Java programs.
- **Compiler (javac):** The javac compiler translates Java source code into bytecode (.class files) that the JVM can run.
- **Java Debugger (jdb):** A tool for debugging Java applications.

- **Other Tools:** Tools like jar (for creating JAR files), javadoc (for generating documentation), and other utility tools that assist in development.

Setting Up the JDK

To develop in Java, you need to install the JDK, which is available from several sources, including Oracle, OpenJDK, Amazon Corretto, and more. Here's a basic setup guide:

1. **Download the JDK** from a trusted provider, like Oracle or OpenJDK.
2. **Install the JDK** following the installer instructions for your operating system.
3. **Configure the Environment Variables**:
    - **JAVA_HOME**: Set this to the JDK installation directory.
    - **Path**: Add JAVA_HOME/bin to your system's PATH so you can run javac and java commands from anywhere.
4. **Verify the Installation**:
    - Run java -version and javac -version in your terminal to ensure they're working.

1.7 The Java Standard Library:

The Java Standard Library, also known as the Java API (Application Programming Interface), is a rich set of prebuilt classes, interfaces, and packages that come bundled with the Java Development Kit (JDK). It provides essential functionalities and tools that help Java developers perform common tasks without writing code from scratch. This library is vast, covering a wide range of utilities and providing a foundation for developing robust applications.

Here's an overview of the main components and packages in the Java Standard Library:

1. Core Packages

- **java.lang**: This is one of the most essential packages in the library, automatically imported by default in every Java program. It includes:
    - Basic language classes like Object, Class, and System.
    - Wrapper classes for primitive types (Integer, Double, Boolean, etc.).
    - Math utilities (Math, StrictMath).
    - String manipulation (String, StringBuilder, StringBuffer).
    - Threading support (Thread, Runnable).
- **java.util**: This package provides utilities for data manipulation, collections, and other functionalities.
    - Collection framework classes like ArrayList, LinkedList, HashMap, HashSet, Queue, Deque.
    - Utility classes like Date, Calendar, Random, UUID, Timer, and Optional.
    - Classes for concurrent programming like ConcurrentHashMap, CopyOnWriteArrayList, ForkJoinPool.
- **java.io**: Provides classes for system input and output, including file handling, serialization, and data streams.

    - Basic stream classes like InputStream, OutputStream, Reader, and Writer.
    - File handling classes like File, FileInputStream, FileReader, BufferedReader, BufferedWriter.
    - Serialization classes (Serializable, ObjectInputStream, ObjectOutputStream).
- **java.nio**: A package offering the New Input/Output (NIO) API for more efficient and flexible I/O operations.
    - Buffer-based data handling with classes like ByteBuffer, CharBuffer.
    - Channels for data streams (FileChannel, SocketChannel).
    - File operations and path handling through java.nio.file.
- **java.net**: Provides classes for networking, allowing programs to communicate over the Internet.
    - Classes for URLs and HTTP communication (URL, URLConnection, HttpURLConnection).
    - Socket programming classes (Socket, ServerSocket, DatagramSocket).
- **java.time**: Introduced in Java 8 to modernize date and time manipulation.
    - Core classes like LocalDate, LocalTime, LocalDateTime, ZonedDateTime.
    - Utilities for period and duration handling (Period, Duration).
    - Formatting classes (DateTimeFormatter, Temporal).

2. Collections Framework (java.util)

Java's Collections Framework, found primarily in java.util, is a powerful set of interfaces and classes for storing and manipulating groups of data. This framework includes several types of collections:

- **Lists**: Ordered collections of elements (ArrayList, LinkedList).
- **Sets**: Unordered collections of unique elements (HashSet, TreeSet).
- **Maps**: Key-value pairs (HashMap, TreeMap, LinkedHashMap).
- **Queues**: Supports FIFO (First-In-First-Out) behavior (PriorityQueue, ArrayDeque).
- **Concurrent Collections**: Thread-safe collections designed for concurrent access (ConcurrentHashMap, CopyOnWriteArrayList).

3. Concurrency Utilities (java.util.concurrent)

45 This package includes classes that help manage multi-threaded programming:

- Executors for managing thread pools (Executor, ExecutorService, ScheduledExecutorService).
- Synchronization aids like CountDownLatch, CyclicBarrier, Semaphore.
- Locks and atomic classes (Lock, ReentrantLock, AtomicInteger, AtomicReference).

4. Input and Output (java.io and java.nio)

Java’s I/O and NIO packages facilitate handling of data input and output through various streams.

- **File I/O**: Basic file operations are supported by File and FileInputStream classes in java.io, while java.nio.file provides more advanced capabilities.
- **Data Streams**: Stream-based classes for byte data (InputStream, OutputStream) and character data (Reader, Writer).
- **Serialization**: Allows saving objects to a file or transferring them over a network using Serializable.

5. Networking (java.net)

Java's networking package allows programs to communicate across networks, supporting both low-level socket programming and higher-level URL-based communication.

- **URL and URI**: Classes to represent resources (URL, URI).
- **Sockets**: Low-level network connections (Socket, ServerSocket).
- **HTTP and Other Protocols**: Classes to make HTTP requests (HttpURLConnection, HttpClient in newer Java versions).

6. JavaFX and AWT/Swing (GUI Libraries)

- **java.awt** and **javax.swing**: Java's original GUI libraries for creating graphical interfaces.
  - Core components for building GUIs (JFrame, JButton, JPanel).
  - Layout managers for arranging components (BorderLayout, GridLayout).
- **JavaFX**: A modern alternative to Swing, providing richer UI elements and media handling capabilities.
  - Classes like Stage, Scene, Button, and TextField for UI creation.
  - Animation and multimedia support (Media, MediaView).

7. Reflection (java.lang.reflect)

This package enables introspection of classes and objects at runtime.

- Classes such as Class, Field, Method, and Constructor allow access to class metadata.
- Reflection supports dynamic method invocation, field access, and creating instances at runtime.

8. Security and Cryptography (java.security)

Java's security package provides a foundation for secure data handling and encryption.

- **Authentication and Authorization**: Classes like Permission, Policy, SecurityManager.
- **Cryptography**: Support for encryption, hashing, and certificates (MessageDigest, KeyFactory, Signature).
- **SSL/TLS**: Tools for secure network communication (SSLContext, TrustManager, KeyManager).

9. Logging (java.util.logging)

Java's built-in logging framework allows for creating logs at different levels (INFO, WARNING, SEVERE, etc.) and sending them to different outputs.

- **Logger**: The primary class for creating logs.
- **Handlers**: Classes for log output (ConsoleHandler, FileHandler).
- **Formatters**: To format log output (SimpleFormatter, XMLFormatter).

10. Java Management Extensions (JMX)

javax.management: Provides tools for managing and monitoring applications.

- Classes like MBeanServer, ObjectName, MBeanServerFactory.
- Allows for the creation and management of MBeans (Managed Beans) for monitoring resources.

11. Database Connectivity (JDBC - java.sql and javax.sql)

Java's JDBC API is used to connect to databases and execute SQL queries.

- **Core Classes**: Connection, Statement, PreparedStatement, ResultSet.
- **Data Source Management**: Managing connections and pooling (DataSource, ConnectionPoolDataSource).

12. Annotations (java.lang.annotation)

Annotations provide metadata for code elements, often used for configuration or by frameworks.

- Built-in annotations like @Override, @Deprecated, @FunctionalInterface.
- Meta-annotations to create custom annotations.

Unit Summary:

Java's origin story is a testament to the power of adaptable, forward-thinking design. From a small project for embedded devices to a language that revolutionized enterprise computing, Java's history showcases the importance of innovation in technology.

- The **JRE** allows you to run Java applications.
- The **JDK** includes the JRE plus tools to compile, debug, and develop Java applications.

Having both the JRE and JDK components is essential if you plan on both developing and running Java applications.

The Java Standard Library is constantly evolving, with new classes and packages added in each Java version to enhance its capabilities and keep up with modern programming needs. It is one of the most comprehensive libraries available, making Java an excellent language for developing a wide variety of applications.

8.1 Check your progress

Here are 20 multiple-choice questions (MCQs) based on the content provided:

---

1. Who developed Java and where?

a) Bjarne Stroustrup at Bell Labs
b) James Gosling at Sun Microsystems
c) Dennis Ritchie at AT&T
d) Guido van Rossum at Google
Answer: b) James Gosling at Sun Microsystems

---

2. What was the initial project name for Java?

a) Green Project
b) Blue Project
c) Sun Project
d) Java Beans
Answer: a) Green Project

---

3. Why was Java initially created?

a) For mobile app development
b) For interactive television and digital devices
c) For creating enterprise applications
d) For web development
Answer: b) For interactive television and digital devices

---

4. What does "WORA" stand for in Java?

a) Write Once, Run Anywhere
b) Write Often, Run Anywhere

c) Write Once, Reuse Always
d) Write Once, Read Anywhere
Answer: a) Write Once, Run Anywhere

---

5. Which feature of Java ensures platform independence?

a) Garbage collection
b) Bytecode and JVM
c) Class inheritance
d) Java Development Kit
Answer: b) Bytecode and JVM

---

6. Which principle is NOT part of Java's design philosophy?

a) Simplicity and Ease of Use
b) Low-Level Memory Access
c) Robustness and Security
d) Dynamic and Versatile
Answer: b) Low-Level Memory Access

---

7. Which Java OOP principle promotes hiding implementation details?

a) Inheritance
b) Polymorphism
c) Encapsulation
d) Abstraction
Answer: d) Abstraction

---

8. What keyword is used to inherit a class in Java?

a) implement
b) extend
c) inherit
d) derive
Answer: b) extend

---

9. Which of the following is NOT an OOP concept in Java?

a) Encapsulation
b) Polymorphism

c) Multithreading
d) Inheritance
Answer: c) Multithreading

---

10. What is the purpose of the JVM?

a) Compiling Java code to bytecode
b) Running Java bytecode on any platform
c) Generating Javadoc files
d) Debugging Java programs
Answer: b) Running Java bytecode on any platform

---

11. Java applets run in a ___ environment.

a) sandbox
b) unrestricted
c) system shell
d) dedicated server
Answer: a) sandbox

---

12. Which statement about Java applications is true?

a) They require a web browser to execute.
b) They operate in a restricted environment.
c) They execute as standalone programs.
d) They are obsolete like applets.
Answer: c) They execute as standalone programs.

---

13. What is included in the JDK but NOT in the JRE?

a) Java Virtual Machine (JVM)
b) Java libraries (like java.util)
c) Compiler and debugger tools
d) Java Runtime Environment (JRE)
Answer: c) Compiler and debugger tools

---

14. Which component of the JDK translates source code into bytecode?

a) JVM
b) javac

c) JRE
d) JSL
Answer: b) javac

---

15. Which Java package is automatically imported by default?

a) java.util
b) java.io
c) java.net
d) java.lang
Answer: d) java.lang

---

16. Which package provides classes for collections in Java?

a) java.lang
b) java.util
c) java.nio
d) java.net
Answer: b) java.util

---

17. What class is used to handle file operations in Java?

a) FileHandler
b) FileReader
c) File
d) FileWriter
Answer: c) File

---

18. Which feature of Java allows concurrent thread execution?

a) Encapsulation
b) Multithreading
c) Abstraction
d) Serialization
Answer: b) Multithreading

---

19. What does JSL stand for in the context of Java?

a) Java System Library
b) Java Standard Library

c) Java Software Logic
d) Java Script Library
Answer: b) Java Standard Library

---

20. Which Java framework is commonly used for enterprise-level applications?

a) Swing
b) Spring
c) JavaFX
d) AWT
Answer: b) Spring

Q. fill in the blanks 1

1. Java was originally developed in the early 1990s by a team led by _____ at Sun Microsystems.
2. The project under which Java was initially developed was called the _____ Project.
3. Java's philosophy "Write Once, Run Anywhere" (WORA) is made possible by compiling Java code into _____.
4. In Java, the process of hiding complex implementation details and exposing only essential features is called _____.
5. Java applets typically run within a _____ and are embedded in HTML.
6. The Java Development Kit (JDK) includes the JRE and additional tools like the _____, which translates Java source code into bytecode.
7. The _____ package in the Java Standard Library is automatically imported and includes classes like Object, String, and Math.
8. The _____ framework in the java.util package provides utilities like ArrayList, HashMap, and LinkedList for managing groups of data.
9. The _____ API allows Java programs to connect to databases and execute SQL queries.
10. _____ was introduced in Java 8 to provide modern tools for handling dates and times, replacing the older java.util.Date class.

Q3. Answer the following:

1) **Who led the team that developed Java, and at which company?**
2) **What was the initial name of the Java programming language, and why was it changed?**
3) **Explain the concept of "Write Once, Run Anywhere" (WORA) in Java.**
4) **What is the main difference between Java Applets and Java Applications in terms of deployment?**
5) **List two core principles of Java's object-oriented programming features.**
6) **What is the role of the Java Virtual Machine (JVM) in the Java environment?**

7) **Name two tools included in the Java Development Kit (JDK) apart from the JRE.**
8) **Which Java package provides classes for network programming, such as handling URLs and sockets?**
9) **What is the purpose of the `java.util.logging` package in Java?**
10) **How does Java achieve security for its applications, especially applets?**

Unit 2: **Java language fundamentals**

2.1 Introduction
2.2 Lexical units, Keywords, variables, constants vs variables
2.3 The scope and lifetime of variable,
2.4 Data types, Type conversion: Coercion and casting
2.5 Operators & expressions, precedence of operators
2.6 Control statements
2.7 Practical examples
2.8 Arrays.
2.9 Unit summery
2.10 Check your progress

2.1 Introduction:

Together, the elements **Lexical units, Keywords, variables and constants** form the foundation of Java syntax and are crucial for writing functional and error-free code.

2.2 Lexical units, Keywords, variables, constants:

In Java, **lexical units** are the basic building blocks of a program, including keywords, literals, identifiers, operators, and punctuators/delimiters. They define the structure and syntax of the code, enabling Java to interpret and execute it correctly.

Keywords are reserved words in Java that have special meanings and cannot be used for naming variables or methods. Examples include class, public, static, if, and while. Keywords are essential for defining the program structure and executing control flow.

Variables in Java are named storage locations that hold data values. Each variable must have a specific data type, like int, double, or String, defining the type of data it can store. Variables are essential for storing and manipulating data throughout a Java program, and they must follow naming conventions and rules (e.g., starting with a letter and avoiding keywords).

The latest version of Java, Java 21, includes a set of reserved keywords, which are integral to the language and cannot be used as identifiers (variable names, method names, etc.). Here's an overview of the key reserved keywords in Java, which remain consistent with earlier versions, with a few notable updates in the language's syntax and features:

Core Keywords (unchanged from previous versions):

- **Access Modifiers**: public, protected, private
- **Primitive Types**: int, byte, short, long, float, double, boolean, char
- **Control Flow**: if, else, switch, case, default, do, while, for, continue, break, return
- **Exception Handling**: try, catch, finally, throw, throws
- **Object-Oriented**: class, interface, extends, implements, new, this, super, instanceof

- **Miscellaneous**: abstract, static, final, void, volatile, synchronized, transient

New Language Features in Java 21:

While these keywords remain, Java 21 introduces advanced features that refine the language:

1. **Record Patterns**: Allows for concise data structure matching and destructuring in pattern matching.
2. **Pattern Matching for switch**: Extends pattern matching to switch statements, enhancing control flow based on type matching.
3. **Unnamed Variables and Patterns**: You can use _ as an unnamed variable, useful for cases where values are ignored.
4. **String Templates (Preview)**: Java 21 allows for embedded expressions in strings, simplifying formatting (similar to templating in other languages).
5. **Scoped Values (Preview)**: This feature offers a cleaner way to define values limited in scope, particularly useful in multithreading contexts.

These improvements build on the existing keywords while adding syntactical flexibility without introducing new keywords per se.

For a complete list and more on Java 21's syntax and features, you can visit the official Oracle Docs:

tps://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html).

Here is a chronological list of Java versions along with their release years:

1. **JDK 1.0** – January 1996
2. **JDK 1.1** – February 1997
3. **J2SE 1.2** (Java 2) – December 1998
4. **J2SE 1.3** – May 2000
5. **J2SE 1.4** – February 2002
6. **J2SE 5.0** (also known as Java 5) – September 2004
7. **Java SE 6** – December 2006
8. **Java SE 7** – July 2011
9. **Java SE 8** – March 2014
10. **Java SE 9** – September 2017
11. **Java SE 10** – March 2018
12. **Java SE 11** (LTS) – September 2018
13. **Java SE 12** – March 2019
14. **Java SE 13** – September 2019
15. **Java SE 14** – March 2020
16. **Java SE 15** – September 2020
17. **Java SE 16** – March 2021
18. **Java SE 17** (LTS) – September 2021
19. **Java SE 18** – March 2022

20. **Java SE 19** – September 2022
21. **Java SE 20** – March 2023
22. **Java SE 21** (LTS) – September 2023

The "LTS" (Long-Term Support) versions receive extended support and are commonly used in production environments. The other versions receive updates for six months only, as Oracle now releases new Java versions every six months.

constants and **variables** in Java:

In Java, **constants** and **variables** both represent values, but they are used differently in terms of mutability and are declared differently. Here's a breakdown of their key differences:

Variables:

- **Definition**: Variables are named storage locations that hold data that can change throughout the execution of a program.
- **Syntax**: Declared with a data type (e.g., int, double, String) followed by the variable name.
- **Example**:

```
int age = 25;
String name = "John";
age = 30; // Variable 'age' value is changed
```

- **Usage**: Variables are mutable, meaning their values can be updated throughout the code. They are used to store values that might change, such as counters, user inputs, or intermediate results.

Constants:

- **Definition**: Constants are variables whose values are set once and cannot change after they are assigned. They represent fixed values throughout the program's execution.
- **Syntax**: Declared with the final keyword and usually static if shared across instances of a class.
- **Naming Convention**: Constants are typically written in all uppercase letters with words separated by underscores (e.g., MAX_SPEED).
- **Example**:

```
final int MAX_SPEED = 120;
final double PI = 3.14159;
```

- **Usage**: Constants are immutable and are used to represent values that should not change, like physical constants, configuration settings, or fixed limits.

Key Differences:

Feature	Variables	Constants
Mutability	Mutable (values can change)	Immutable (values cannot change)
Declaration	Just the data type and name	Use final keyword
Naming	Follows variable naming	Typically all uppercase with

Feature	Variables	Constants
	conventions	underscores
Example	int age = 25;	final int MAX_SPEED = 120;

Practical Tips:

- Use constants for values that represent fixed data and shouldn't change, as this helps prevent accidental modifications.
- Constants can make your code more readable and maintainable; as they clarify that certain values are unchangeable.

2.3 The scope and lifetime of variable:

In Java, the *scope* and *lifetime* of a variable determine where the variable can be accessed and how long it exists in memory. Here's a breakdown of both concepts:

Scope of a Variable

The scope of a variable in Java defines the parts of the program where the variable can be accessed. Java has several kinds of variable scopes:

Local Variables:

- Declared within a method, constructor, or block.
- Accessible only within that method, constructor, or block.
- Not accessible outside of this scope, so you cannot access a local variable from other methods.

```
void myMethod() {
  int localVar = 10; // localVar is only accessible within myMethod
  }
```

Instance Variables (also called *non-static fields*):

- Declared inside a class but outside any method or constructor, without the static keyword.
- Associated with an instance of the class, so each object has its own copy.
- Accessible from any non-static method of the class, but scope is limited to the class itself.

```
public class MyClass {
        int instanceVar; // instanceVar can be accessed in any method within MyClass
        }
```

Class Variables (also called *static fields*):

- Declared with the static keyword inside a class but outside any method or constructor.
- Shared across all instances of the class; there is only one copy of a static variable.
- Accessible throughout the class, and even outside the class if the access modifier allows.

```
public class MyClass {
```

```
    static int classVar; // classVar can be accessed by all instances and static methods
}
```

Block Variables:

- Declared within blocks such as loops, conditionals, or other code blocks within a method.
- Scope is limited to that specific block; once the block is exited, the variable goes out of scope.

```
void myMethod() {
    for (int i = 0; i < 5; i++) {
        int blockVar = i * 2; // blockVar's scope is limited to this loop
} }
```

Lifetime of a Variable

The lifetime of a variable refers to how long the variable occupies memory during program execution.

Local Variables:

- Exist only during the execution of the method or block where they are declared.
- Once the method or block completes, the memory for local variables is reclaimed.

Instance Variables:

- Exist as long as the instance of the class exists.
- When the object is no longer reachable (e.g., no more references to it exist), it becomes eligible for garbage collection, and its instance variables are reclaimed.

Class Variables:

- Exist as long as the class is loaded in the JVM.
- They are created when the class is loaded and reclaimed when the class is unloaded or when the application terminates.

Block Variables:

- Have the same lifetime as local variables.
- Their memory is freed once the block where they were declared is exited.

Example Illustrating Scope and Lifetime

```
public class Example {
    static int classVar = 0; // Class variable: scope is the entire class,
                             // lifetime is as long as class is loaded
    int instanceVar = 10; // Instance variable: scope is entire class,
                          // lifetime is as long as object exists
    void someMethod() {
        int localVar = 5; // Local variable: scope is someMethod,
```

```
                    // lifetime is only within this method call
        for (int i = 0; i < 3; i++) {
                    int blockVar = i * 2;
                    // Block variable: scope is this for-loop,
                    // lifetime is within each loop iteration
                    System.out.println(blockVar);
                    }
                    // blockVar is no longer accessible here
                    }
        }
```

In this example:

- classVar can be accessed anywhere in Example and exists for the program's duration.
- instanceVar exists as long as the object of Example exists.
- localVar is created each time someMethod is called and removed when the method exits.
- blockVar is created and removed within each loop iteration.

2.3 Data types

In Java, data types are broadly divided into **primitive** data types and **non-primitive** (or reference) data types. Each type serves different purposes and has specific features. Let's look at each category.

1. Primitive Data Types

These are the most basic data types built into the Java language.

There are **8 primitive data types**:

1. **byte**
   - Size: 1 byte (8 bits)
   - Range: -128 to 127
   - Used to save space in large arrays.
2. **short**
   - Size: 2 bytes (16 bits)
   - Range: -32,768 to 32,767
   - Useful for saving memory in large arrays when the byte type is too small.
3. **int**

- Size: 4 bytes (32 bits)
- Range: -2^31 to 2^31 - 1 (about -2 billion to 2 billion)
- Default choice for integral values unless there's a reason to use a smaller data type.

4. **`long`**

- Size: 8 bytes (64 bits)
- Range: -2^63 to 2^63 - 1
- Used when a wider range than `int` is needed.
- You should use an "L" suffix to specify a `long` literal, e.g., `1234L`.

5. **`float`**

- Size: 4 bytes (32 bits)
- Precision: Approximately 6-7 decimal digits
- Used for fractional values with limited precision.
- Suffix "f" is used to specify a `float` literal, e.g., `1.23f`.

6. **`double`**

- Size: 8 bytes (64 bits)
- Precision: Approximately 15-16 decimal digits
- Default choice for decimal values.
- Has better precision than `float`.

7. **`char`**

- Size: 2 bytes (16 bits)
- Range: 0 to 65,535 (Unicode characters)
- Used to store a single character, like `'a'` or `'%'`.

8. **`boolean`**

- Size: Approximately 1 bit (though the exact storage can vary)
- Values: `true` or `false`
- Used for true/false conditions.

2. Non-Primitive (Reference) Data Types

These types are not defined by Java itself but by the user. Non-primitive data types are **objects** or **references to objects**. Some common types include:

1. **`Strings`**

- A sequence of characters, e.g., `"Hello World"`.
- Defined by the `String` class.

2. **`Arrays`**

- Collection of elements of the same data type.

- E.g., `int[] arr = {1, 2, 3};`

3. Classes

- Defines custom data types by grouping fields (data) and methods (functions).
- Used to create objects.

4. Interfaces

- Abstract types used to specify behaviors that classes must implement.

5. Other Types

- Includes various classes, enumerations (enums), and collections like `List`, `Set`, `Map`, etc.

Summary Table

Category	Data Type	Size (bits)	Range / Description
Primitive	byte	8	-128 to 127
	short	16	-32,768 to 32,767
	int	32	-2^31 to 2^31 - 1
	long	64	-2^63 to 2^63 - 1
	float	32	6-7 decimal digits precision
	double	64	15-16 decimal digits precision
	char	16	0 to 65,535 (Unicode)
	boolean	~1 bit	true or false
Non-Primitive	String	Varies	Sequence of characters
	Array	Varies	Collection of elements
	Class/Object	Varies	Custom data structures
	Interface	Varies	Specifies behaviors

Each of these data types has specific uses and behaviors, allowing for efficient memory management and better organization in Java applications.

2.4 Type conversion:

In Java, type conversion refers to the process of converting data from one type to another, often to perform operations or handle specific types of data requirements. This can be divided into two main categories:

1. **Coercion (Automatic/Implicit Conversion)**
2. **Casting (Explicit Conversion)**

1. Coercion (Automatic/Implicit Conversion)

- **Definition**: Coercion happens automatically when the Java compiler converts one type to another without explicit instruction from the programmer.
- **When it Happens**: Coercion generally occurs when converting a smaller type to a larger type, where there is no risk of data loss.
- **Examples of Coercion**:
  - Converting an int to a long.
  - Converting a float to a double.

```
int myInt = 100;
long myLong = myInt; // int is coerced to long automatically
```

Since long has a larger storage size than int, the conversion is safe and does not require explicit casting.

2. Casting (Explicit Conversion)

- **Definition**: Casting is a form of explicit conversion where the programmer specifies the target type. This type of conversion can be risky because data might be lost, especially when converting from a larger type to a smaller type.
- **When it Happens**: Casting is needed when going from a larger type to a smaller type or when changing types that aren't compatible implicitly.
- **Examples of Casting**:
  - Converting a double to an int (may lead to data loss due to truncation of decimal places).
  - Converting from int to byte (may lead to overflow if the int value is larger than the range of byte).

```
double myDouble = 10.5;
int myInt = (int) myDouble; // Casting double to int explicitly
```

Here, the decimal portion of myDouble (0.5) will be truncated, so myInt will have the value 10.

Key Differences between Coercion and Casting:

Feature	Coercion (Automatic Conversion)	Casting (Explicit Conversion)
Initiated by	Compiler	Programmer
Risk of Data Loss	Minimal (usually no risk)	Possible risk of data loss
Syntax	No special syntax required	Explicit syntax required (type)
Examples	int to long, float to double	double to int, int to byte

In summary, coercion is automatic and low-risk, suitable for widening conversions, while casting is explicit, potentially risky, and often used in narrowing conversions.

2.5 Operators & expressions, precedence of operators

In Java, operators and expressions are fundamental for performing calculations, making decisions, and manipulating data. Here's a rundown of the types of operators in Java, how expressions work, and the rules of operator precedence.

1. Types of Operators

Java provides various types of operators, including:

Arithmetic Operators

These are used for basic mathematical operations:

- + Addition
- - Subtraction
- * Multiplication
- / Division
- % Modulus (remainder)

Unary Operators

These operate on a single operand:

- + Unary plus (indicates a positive value)
- - Unary minus (negates an expression)
- ++ Increment (increases the value by 1)
- -- Decrement (decreases the value by 1)
- ! Logical complement (negates a boolean value)

Relational Operators

These are used to compare two values:

- == Equal to
- != Not equal to
- > Greater than
- < Less than
- >= Greater than or equal to
- <= Less than or equal to

Logical Operators

These work with boolean values:

- && Logical AND

- || Logical OR
- ! Logical NOT

Bitwise Operators

These operate on bits within an integer type:

- & Bitwise AND
- | Bitwise OR
- ^ Bitwise XOR
- ~ Bitwise complement
- << Left shift
- >> Right shift
- >>> Unsigned right shift

Assignment Operators

These assign values to variables:

- = Simple assignment
- += Add and assign
- -= Subtract and assign
- *= Multiply and assign
- /= Divide and assign
- %= Modulus and assign

Conditional (Ternary) Operator

The only ternary operator in Java is used as a shorthand for if-else:

- ?: (e.g., condition ? expression1 : expression2)

Instanceof Operator

This checks if an object is an instance of a specific class or subclass:

- instanceof

2. Expressions in Java

An **expression** is a construct made up of variables, operators, and method calls that are evaluated to produce a single value. For example:

int result = (10 + 5) * 3; // Here, (10 + 5) * 3 is an expression

3. Precedence and Associativity of Operators

Operators in Java follow specific precedence and associativity rules that determine the order in which parts of an expression are evaluated.

- **Precedence** dictates which operator has priority when evaluating expressions.
- **Associativity** defines the order in which operators of the same precedence are processed.

Java Operator Precedence Table

From highest to lowest precedence:

Precedence	Operators	Description	Associativity
1	()	Parentheses	Left to Right
2	++ --	Postfix increment/decrement	Left to Right
3	++ -- + - ~ !	Unary operators	Right to Left
4	* / %	Multiplication, division, modulus	Left to Right
5	+ -	Addition, subtraction	Left to Right
6	<< >> >>>	Bitwise shift	Left to Right
7	< <= > >=	Relational	Left to Right
8	== !=	Equality	Left to Right
9	&	Bitwise AND	Left to Right
10	^	Bitwise XOR	Left to Right
11	`	`	Bitwise OR
12	&&	Logical AND	Left to Right
13	`		`
14	?:	Ternary	Right to Left
15	= += -= *= /= %= &= `	= ^= <<= >>= >>>=`	Assignment

Example of Precedence and Associativity in Action

Consider this expression: int result = 5 + 2 * 3 - 4 / 2;

1. **Multiplication** (*) and **division** (/) have higher precedence than **addition** (+) and **subtraction** (-), so they are evaluated first.
   - 2 * 3 = 6
   - 4 / 2 = 2
2. After that, the expression is: int result = 5 + 6 - 2;
3. Now, **addition** and **subtraction** are evaluated from left to right.
   - 5 + 6 = 11

- 11 - 2 = 9

So, result will be 9.

2.6 Control statements:

Control statements in Java allow you to manage the flow of execution in a program based on conditions or loops. There are three primary types of control statements:

1. **Decision-Making Statements:**
   These statements execute specific code blocks based on given conditions.
   - if statement: Executes code if a condition is true.
   - if-else statement: Executes one block if the condition is true, otherwise another block.
   - else-if ladder: Allows multiple conditions to be checked sequentially.
   - switch statement: Executes one of many possible code blocks based on the value of an expression.

   Example:

```
int num = 10;

// if-else statement
if (num > 0) {
   System.out.println("Positive number");
} else {
   System.out.println("Negative number");
}

// switch statement
switch (num) {
   case 1:
      System.out.println("One");
      break;
   case 10:
      System.out.println("Ten");
      break;
   default:
      System.out.println("Unknown number");
}
```

2. **Looping Statements:**
   These statements repeat a block of code while a condition is true.
   - for loop: Runs a loop a fixed number of times.
   - while loop: Repeats while a condition is true.
   - do-while loop: Similar to while, but runs at least once.

Example:

```
int i = 0;

// for loop
for (int j = 0; j < 5; j++) {
   System.out.println("for loop: " + j);
}

// while loop
while (i < 5) {
   System.out.println("while loop: " + i);
   i++;
}

// do-while loop
do {
   System.out.println("do-while loop: " + i);
   i--;
} while (i > 0);
```

3. **Jump Statements:**
   These control the flow of execution by jumping to a different part of the code.
   - break: Exits a loop or switch statement.
   - continue: Skips the current iteration and proceeds to the next loop iteration.
   - return: Exits a method and optionally returns a value.

   Example:

```
for (int k = 0; k < 10; k++) {
   if (k == 5) break;  // exit the loop when k is 5
   if (k % 2 == 0) continue;  // skip even numbers
   System.out.println(k);
}
```

These control statements allow Java developers to make programs more dynamic and responsive based on input or conditions.

2.8 Practical examples:

a. Selection stetements:

Example 1: if statement

```
public class IfExample {
```

```
    public static void main(String[] args) {
        int age = 20;

        if (age >= 18) {
            System.out.println("You are eligible to vote.");
        }
    }
}
```

Example 2: if-else statement

```
public class IfElseExample {
    public static void main(String[] args) {
        int marks = 45;

        if (marks >= 50) {
            System.out.println("You passed the exam.");
        } else {
            System.out.println("You failed the exam.");
        }
    }
}
```

Example 3: if-else-if ladder

```
public class IfElseIfExample {
    public static void main(String[] args) {
```

```
        int marks = 85;

        if (marks >= 90) {
            System.out.println("Grade: A+");
        } else if (marks >= 80) {
            System.out.println("Grade: A");
        } else if (marks >= 70) {
            System.out.println("Grade: B");
        } else if (marks >= 60) {
            System.out.println("Grade: C");
        } else {
            System.out.println("Grade: F");
        }
    }
}
```

Example 4: Nested if

```
public class NestedIfExample {
    public static void main(String[] args) {
        int age = 25;
        boolean hasID = true;

        if (age >= 18) {
            if (hasID) {
```

```
            System.out.println("You can enter the club.");
        } else {
            System.out.println("You need an ID to enter.");
        }
    } else {
        System.out.println("You are not old enough to enter.");
    }
}
```

Note:

- if: Simple condition check.
- if-else: Handles two outcomes.
- if-else-if: Handles multiple exclusive conditions.
- Nested if: Adds complexity by checking conditions within other conditions.

b. loops: for, while, do-while

Example 1: Simple for loop

```
public class ForLoopExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            System.out.println("Number: " + i);
        }
    }
}
```

Example 2: Nested for loop

```
public class NestedForLoopExample {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            for (int j = 1; j <= 5; j++) {
                System.out.print(i * j + "\t");
            }
            System.out.println(); // Move to the next line
        }
    }
```

```
}
```

Example 3: while loop

```
public class WhileLoopExample {
    public static void main(String[] args) {
        int count = 10;
        while (count > 0) {
            System.out.println("Count: " + count);
            count--;
        }
    }
}
```

Example 4: Nesting of while and for loop

```
public class TrianglePattern {
    public static void main(String[] args) {
        int rows = 5;
        int currentRow = 1;

        while (currentRow <= rows) {
            for (int j = 1; j <= currentRow; j++) {
                System.out.print("*");
            }
            System.out.println(); // Move to the next line
            currentRow++;
        }
    }
}
```

Example 5: do-while loop with switch-case statement

```
import java.util.Scanner;

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

        do {
            System.out.println("Menu:");
            System.out.println("1. Option 1");
            System.out.println("2. Option 2");
            System.out.println("3. Exit");
            System.out.print("Enter your choice: ");
            choice = scanner.nextInt();
```

```
            switch (choice) {
                case 1 -> System.out.println("You chose Option 1.");
                case 2 -> System.out.println("You chose Option 2.");
                case 3 -> System.out.println("Exiting...");
                default -> System.out.println("Invalid choice. Try again.");
            }
        } while (choice != 3);

        scanner.close();
    }
}
```

Check your progress 4

Multiple-Choice Questions (MCQs)

1. Which of the following is a primitive data type in Java? a) String
   b) int
   c) Array
   d) Object
2. What is the range of a byte in Java? a) -32768 to 32767
   b) -128 to 127
   c) -2147483648 to 2147483647
   d) 0 to 255
3. Which operator is used for logical AND in Java? a) ||
   b) &&
   c) &
   d) ^
4. Which of the following is used to declare a floating-point literal in Java? a) 1234d
   b) 1234f
   c) 1234.0
   d) 1234L
5. What is the default value of a boolean variable in Java? a) 0
   b) null
   c) false
   d) true
6. Which of the following is an example of coercion in Java? a) Converting int to long
   b) Converting double to int
   c) Converting int to byte
   d) Converting char to string
7. In Java, which data type is used for storing single characters? a) String
   b) char
   c) int
   d) boolean

8. Which loop is guaranteed to execute at least once in Java? a) for loop
   b) while loop
   c) do-while loop
   d) none of the above
9. What is the range of a short data type in Java? a) -32768 to 32767
   b) -128 to 127
   c) -2^31 to 2^31 - 1
   d) 0 to 65535
10. Which of the following is true about type casting in Java? a) It always happens automatically
    b) It only works for primitive data types
    c) It can convert from larger to smaller types without risk
    d) It requires explicit syntax for smaller to larger conversions

Fill-in-the-Blank Questions

1. The size of a `long` data type in Java is _____ bytes.
2. The ____ operator is used to check if two objects are equal in Java.
3. In Java, the ____ operator is used to increment a value by 1.
4. When converting a `float` to a `double`, the conversion is _____.
5. The default value for an uninitialized `boolean` variable is _____.
6. To specify a `double` literal, you can use the suffix _____.
7. A `for` loop has the general structure of _____.
8. The _____ operator is used to negate a boolean expression in Java.
9. The size of an `int` in Java is _____ bytes.
10. The _____ statement in Java is used to exit a loop or switch statement prematurely.

Short-Answer Questions

1. What are the 8 primitive data types in Java?
2. Explain the difference between coercion and casting in Java.
3. What is the default choice for integral values in Java?
4. What is the significance of the `L` suffix when working with long literals in Java?
5. Write a simple Java program that uses an `if-else` statement to check if a number is positive or negative.
6. What are the three types of control statements in Java?
7. What happens when a `continue` statement is encountered in a loop in Java?
8. How does the precedence of operators affect the evaluation of expressions in Java? Give an example.
9. What is the difference between a `while` loop and a `do-while` loop in Java?
10. Describe how an array is declared and initialized in Java with an example.

2.8 Arrays in Java

1. Introduction to Arrays

In Java, an array is a data structure that can hold a fixed-size collection of elements of the same type. It allows for efficient storage and manipulation of multiple values in a single variable, making it easier to handle large datasets or related pieces of data.

2. Declaring an Array

To declare an array in Java, you define the type of the array elements followed by square brackets [], and then the array variable name.

```
int[] numbers;  // Declaring an array of integers
String[] names; // Declaring an array of strings
```

3. Initializing Arrays

Arrays in Java can be initialized in two primary ways:

1. Static Initialization (at declaration):

```
int[] numbers = {1, 2, 3, 4, 5};
String[] names = {"Alice", "Bob", "Charlie"};
```

2. Dynamic Initialization (using the new keyword):

```
int[] numbers = new int[5];  // Array with 5 elements, default values will be 0
String[] names = new String[3];  // Array with 3 elements, default values will be null
```

4. Accessing Elements in an Array

Array elements are accessed using their index, starting from 0. To access the first element of an array, you use index 0.

```
int firstNumber = numbers[0]; // Accessing the first element
String firstName = names[0];  // Accessing the first name
```

5. Array Length

The length of an array can be accessed through the length property, which tells you how many elements the array contains.

```
int arrayLength = numbers.length;  // Returns the number of elements in the 'numbers' array
```

6. Modifying Array Elements

You can modify the elements of an array by assigning a new value to a specific index.

```
numbers[0] = 10;  // Setting the first element of the array to 10
names[1] = "Eve";  // Changing the second element of the array to "Eve"
```

7. Iterating Through an Array

There are several ways to loop through an array in Java. Here are the most common methods:

1. Using a for loop:

```
for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

2. Using an enhanced for loop (also known as the "foreach" loop):

```
for (String name : names) {
    System.out.println(name);
}
```

8. Multidimensional Arrays

In Java, you can create arrays of arrays, commonly referred to as multidimensional arrays. A two-dimensional array is the most common form.

Declaring a two-dimensional array:

```
int[][] matrix = new int[3][3];  // A 3x3 matrix
```

Initializing a two-dimensional array:

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

Accessing elements in a two-dimensional array:

```
int element = matrix[1][2];  // Accessing the element in the second row, third column (6)
```

9. Array Manipulation

1. Sorting Arrays:

Java provides a utility method Arrays.sort() to sort arrays. It is part of the java.util package.

```
import java.util.Arrays;

Arrays.sort(numbers);  // Sorts the 'numbers' array in ascending order
```

2. Searching in Arrays:

You can use Arrays.binarySearch() to find an element in a sorted array.

```
int index = Arrays.binarySearch(numbers, 5);  // Finds the index of 5 in the sorted array
```

3. Copying Arrays:

You can copy arrays using Arrays.copyOf() or System.arraycopy().

```
int[] copiedArray = Arrays.copyOf(numbers, numbers.length);  // Creates a copy of the 'numbers' array
```

10. ArrayLists vs Arrays

While arrays are powerful and efficient, they have some limitations compared to ArrayList:

- **Fixed Size**: Once an array is created, its size cannot be changed.
- **Type Safety**: Arrays hold elements of a specific type, but arrays do not have the flexibility to work with other types like ArrayList.

However, arrays are still more efficient in terms of memory allocation and speed when working with a known, fixed-size dataset.

Array examples:

1. Finding the Maximum Element in an Array:

```
int max = numbers[0];
for (int i = 1; i < numbers.length; i++) {
    if (numbers[i] > max) {
        max = numbers[i];
    }
}
```

2. Reversing an Array:

```
for (int i = 0; i < numbers.length / 2; i++) {
    int temp = numbers[i];
    numbers[i] = numbers[numbers.length - 1 - i];
```

```
    numbers[numbers.length - 1 - i] = temp;
}
```

3. Summing the Elements of an Array:

```
int sum = 0;
for (int num : numbers) {
    sum += num;
}
```

Conclusion: Arrays are a foundational concept in Java programming, allowing developers to efficiently store and manipulate collections of data. Mastering arrays is critical for tackling complex problems and understanding more advanced data structures in Java. Whether you're building small applications or large systems, arrays will often be the go-to solution for storing ordered data.

Check your progress:

Multiple-Choice Questions (MCQs)

1. **How do you declare an array of integers in Java?**
   - A) `int numbers[];`
   - B) `int[] numbers;`
   - C) `Array<int> numbers;`
   - D) `int[] = numbers;`
2. **What is the default value of an uninitialized integer array in Java?**
   - A) 1
   - B) 0
   - C) null
   - D) Undefined
3. **Which of the following is the correct way to initialize a two-dimensional array with values in Java?**
   - A) `int[][] matrix = {1, 2, 3, 4, 5, 6};`
   - B) `int[][] matrix = {{1, 2, 3}, {4, 5, 6}};`
   - C) `int matrix = new int[3][3];`
   - D) `int matrix = new int[3][3] {1, 2, 3, 4, 5, 6};`
4. **What method is used to sort an array in Java?**
   - A) `Arrays.sort()`
   - B) `Arrays.order()`
   - C) `Array.sort()`
   - D) `ArrayList.sort()`
5. **Which statement about arrays in Java is true?**
   - A) Arrays are dynamic in size.
   - B) Arrays can hold elements of different data types.
   - C) Array elements are accessed using indices starting from 0.
   - D) Arrays cannot be modified once created.

[Answers: 1. B) `int[] numbers;` **2.** B) 0

3. `int[][] matrix = {{1, 2, 3}, {4, 5, 6}};` 4. A) `Arrays.sort()`

5. Array elements are accessed using indices starting from 0.]

True/False Questions

1. In Java, the length of an array can be modified after the array is created.
2. You can access the first element of an array using index 0.
3. The default value of an uninitialized String array in Java is null.
4. A two-dimensional array in Java can be declared as **int[][] matrix = new int[3][3];**.
5. ArrayLists in Java are always more efficient than arrays in terms of memory usage and speed. **[Answers: False, True, True, True, False]**

These questions cover key concepts related to arrays in Java, such as declaration, initialization, accessing elements, array manipulation, and differences between arrays and ArrayLists.

2.9 Summary:

This unit explains crucial elements of Java program such as Lexical units, Keywords, variables, constants vs variables, the scope and lifetime of variable. Along with this fundamental concepts of Data types, Type conversion: Coercion and casting, Operators & expressions, precedence of operators are discussed with examples. Finally advance concepts of Control statements and arrays are discussed.

2.9 Unit summary

Unit Summary

Lexical Units, Keywords, Variables, Constants vs Variables

- **Lexical Units:** The basic building blocks of a programming language, including tokens like keywords, identifiers, literals, operators, and special symbols.
- **Keywords:** Reserved words with predefined meanings, such as int, if, and return.
- **Variables:** Named storage locations in memory used to store data, whose values can change during program execution.
- **Constants vs Variables:** Constants have fixed values defined during program execution (e.g., const PI = 3.14), whereas variables can change their values.

The Scope and Lifetime of Variables

- **Scope:** Defines the part of the program where a variable is accessible. Scopes can be local (inside a function or block) or global (accessible throughout the program).
- **Lifetime:** The duration for which a variable retains its value in memory. It depends on its storage class (automatic, static, dynamic).

Data Types, Type Conversion: Coercion and Casting

- **Data Types:** Define the type of data a variable can hold, e.g., integers, floats, strings, booleans, etc.
- **Type Conversion:**
  - **Coercion:** Automatic type conversion performed by the compiler (e.g., converting an integer to a float in arithmetic operations).
  - **Casting:** Explicit type conversion specified by the programmer (e.g., (int) 3.14 converts 3.14 to 3).

Operators & Expressions, Precedence of Operators

- **Operators:** Symbols or words that perform operations on operands (e.g., +, -, *, ==, &&).
- **Expressions:** Combinations of variables, constants, and operators that yield a result (e.g., a + b).
- **Precedence of Operators:** Determines the order in which operators are evaluated. For example, multiplication (*) has higher precedence than addition (+).

Control Statements

- **Control Statements:** Instructions that determine the flow of execution in a program.
  - **Conditional Statements:** if, else if, else, switch.
  - **Loops:** for, while, do-while.
  - **Branching Statements:** break, continue, return.

Practical Examples

- Demonstrates the application of the above concepts in real-world programming scenarios.
- Examples include variable declaration, loops for summing arrays, and conditional statements for decision-making.

Arrays

- **Definition:** A collection of elements of the same type, stored in contiguous memory locations, and accessible using indices.
- **Types of Arrays:**
  - One-dimensional (e.g., int arr[5]).
  - Multi-dimensional (e.g., int matrix[3][3] for a 3x3 matrix).
- **Operations:** Initialization, accessing elements, updating values, and iterating through arrays.

2.10 Check your progress

1. **Which of the following is a keyword in Java?** a) variable
   b) static

c) myVariable
d) print

Answer: b) static

2. **What is the primary difference between variables and constants in Java?** a) Variables are mutable; constants are immutable.
   b) Variables cannot be changed; constants can be changed.
   c) Variables use the `final` keyword; constants do not.
   d) Constants can hold more types of data than variables.

   Answer: a) Variables are mutable; constants are immutable.

3. **Which Java keyword is used to define a constant?** a) const
   b) final
   c) static
   d) constant

   Answer: b) final

4. **What is the scope of a local variable in Java?** a) Entire program
   b) Method or block in which it is declared
   c) Class
   d) Instance of the class

   Answer: b) Method or block in which it is declared

5. **Which of the following types of variables are shared across all instances of a class in Java?** a) Local variables
   b) Instance variables
   c) Class variables
   d) Block variables

   Answer: c) Class variables

6. **In Java, which of the following is a control flow keyword?** a) this
   b) for
   c) class
   d) interface

   Answer: b) for

7. **Which data type is used for representing a true or false value in Java?** a) int
   b) char
   c) boolean
   d) String

Answer: c) boolean

8. **What is the lifetime of an instance variable in Java?** a) It exists as long as the class is loaded.
   b) It exists as long as the object is referenced.
   c) It is temporary and only exists within a method call.
   d) It exists as long as the method is executing.

   Answer: b) It exists as long as the object is referenced.

9. **Which keyword is used to create a new object in Java?** a) create
   b) new
   c) this
   d) object

   Answer: b) new

10. **Which of the following is a keyword used in exception handling in Java?** a) try
    b) method
    c) catch
    d) both a and c

    Answer: d) both a and c

Fill in the Blanks

1. In Java, the _____ keyword is used to define a constant whose value cannot be changed after initialization. **Answer**: final
2. A variable that is declared inside a method is called a _____ variable. **Answer**: local
3. _____ variables in Java are shared across all instances of a class and are declared with the `static` keyword. **Answer**: Class
4. _____ variables are associated with an instance of a class, and each object has its own copy. **Answer**: Instance
5. The scope of a _____ variable is limited to the block or loop in which it is defined. **Answer**: block
6. The _____ keyword is used in Java for defining a class method that does not return any value. **Answer**: void
7. In Java, keywords such as `public`, `private`, and `protected` are known as _____ modifiers. **Answer**: access
8. The _____ keyword in Java is used to indicate that a method does not throw any exceptions. **Answer**: throws
9. Java 21 introduced _____, which allows pattern matching for switch statements. **Answer**: Pattern Matching for switch

10. The _____ keyword in Java is used to inherit methods and properties from a parent class. **Answer**: extends

---

Short Answer Questions

1. **What is the purpose of keywords in Java?**
2. **Explain the difference between variables and constants in Java.**
3. **What are instance variables in Java?**
4. **What is the significance of the `final` keyword when declaring a constant in Java?**
5. **Describe the scope and lifetime of a local variable in Java.**
6. **What does the `static` keyword mean when applied to a variable in Java?**
7. **What is the difference between a local variable and a block variable in Java?**
8. **What are the core keywords in Java related to control flow?**
9. **Explain the scope and lifetime of an instance variable in Java.**
10. **What are the new language features introduced in Java 21?**

Unit 3: **Classes and objects:**

Classes and Objects in Java

Introduction: In Java, everything revolves around the concept of *objects*. Java is an object-oriented programming (OOP) language, which means that the programs are based on objects and classes. Understanding how to define classes and create objects is fundamental to writing efficient and maintainable Java code.

In this unit, we'll explore what classes and objects are, how to define them, and how they interact. You'll learn how to create instances of objects from a class and use them to solve problems.

3.1 What is a Class?

A **class** is a blueprint or template for creating objects. It defines properties (fields) and behaviors (methods) of an object created from the given class. A class doesn't occupy memory on its own until it is instantiated into an object (an object is declared/created).

Basic Structure of a Class

Here's a simple class definition in Java:

```
public class Car {
   // Fields (or attributes)
   String color;
   String model;
   int year;

   // Constructor (to initialize the fields)
   public Car(String color, String model, int year) {
      this.color = color;
```

```
        this.model = model;
        this.year = year;
    }

    // Method (or behavior)
    public void startEngine() {
        System.out.println("The engine is now running.");
    }

    public void stopEngine() {
        System.out.println("The engine is now stopped.");
    }
}
```

Explanation:

- **Fields**: These are the attributes that describe the object. For example, in the Car class, the fields color, model, and year store the state of the car.
- **Constructor**: This is a special method used to initialize objects. The constructor is called when an object of the class is created.
- **Methods**: These define the behaviors of an object. The startEngine and stopEngine methods in the Car class represent actions that a car can perform.

3.2 Definition of Object?

An **object** is an instance of a class. When you create an object, you allocate memory for it, and it contains values for the class's fields. The object can use the methods defined in the class. To create an object from the Car class, we use the new keyword, followed by the constructor.

Creating an Object

```
public class Main {
    public static void main(String[] args) {
        // Creating an object of the Car class
        Car myCar = new Car("Red", "Toyota", 2020);

        // Accessing fields of the object
        System.out.println("Car Model: " + myCar.model);
        System.out.println("Car Year: " + myCar.year);
        System.out.println("Car Color: " + myCar.color);

        // Calling methods on the object
        myCar.startEngine();
        myCar.stopEngine();
    }
}
```

Explanation:

- **Instantiation**: new Car("Red", "Toyota", 2020) creates a new object of the Car class.
- **Accessing Object Members**: You can access an object's fields and methods using the dot (.) operator. For example, myCar.model accesses the model field of myCar.

3.3 Constructors in Java

A constructor is a special method used to initialize an object when it is created. The constructor has the same name as the class and does not have a return type. It is automatically called when an object is instantiated.

Default Constructor

If no constructor is provided, Java provides a **default constructor**. It initializes the fields with default values (null for objects, 0 for integers, etc.).

```
public class Car {
    String color;
    String model;
    int year;

    // Default constructor
    public Car() {
        color = "Unknown";
        model = "Unknown";
        year = 0;
    }

    public void startEngine() {
        System.out.println("The engine is now running.");
    }

    public void stopEngine() {
        System.out.println("The engine is now stopped.");
    }
}
```

Note: If you create an object without passing parameters, the default constructor will be invoked:

```
Car myCar = new Car(); // Uses the default constructor
```

3.4 Method Overloading

Java allows you to define multiple methods with the same name, as long as they have different parameter lists. This is known as **method overloading**.

```
public class Car {
   // Method to start the car with a specific gear
   public void startEngine(int gear) {
      System.out.println("Starting engine in gear " + gear);
   }

   // Overloaded method to start the car with a default gear
   public void startEngine() {
      System.out.println("Starting engine in neutral gear");
   }
}
```

In the example above, both methods are named startEngine, but one takes an integer as a parameter, and the other does not. This allows you to use the method in different scenarios.

3.5 Access Modifiers

Access modifiers define the visibility of classes, fields, and methods. The main access modifiers in Java are:

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

Here's an example:

```
public class Car {
   private String color;  // Only accessible within the Car class
   public String model;   // Accessible from any class
   int year;              // Accessible within the same package
}
```

3.6 The this Keyword

The "this" keyword is used inside a method or constructor to refer to the current object. It is commonly used to differentiate between instance variables and parameters when they have the same name.

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

    public Car(String color, String model) {
        this.color = color;    // Refers to the instance variable
        this.model = model;    // Refers to the instance variable
    }
}
```

3.7 The static Keyword

The static keyword is used to indicate that a member (field or method) belongs to the class, rather than to any specific instance of the class. This means that static members are shared by all objects of the class.

Static Fields and Methods

```
public class Car {
    static int numberOfCars = 0;  // Static field shared by all objects

    public Car() {
        numberOfCars++;  // Increment the static field whenever a new object is created
    }

    public static void displayCarCount() {
        System.out.println("Total number of cars: " + numberOfCars);
    }
}
```

You can access the static field and method without creating an object:

```
Car.displayCarCount();  // Access static method
System.out.println(Car.numberOfCars);  // Access static field
```

3.8 Conclusion

In this chapter, you learned how to:

- Define and create classes and objects in Java.
- Use constructors to initialize objects.
- Use methods to define behaviors for objects.
- Work with access modifiers, the this keyword, and static members.

Classes and objects form the foundation of object-oriented programming. Mastering them will enable you to design robust Java applications with clean, reusable code.

Exercises:

1. Create a Book class with fields like title, author, and price. Write methods to display details of the book and apply a discount.
2. Modify the Car class to include a fuelLevel field. Write a method that simulates refueling the car.

3.9 Argument Passing in Java

Java supports **pass-by-value** semantics. For primitive types (e.g., int, double), the actual value is passed. For objects, the reference (or memory address) is passed by value, which allows the method to operate on the original object.

Key Points:

- **Primitive types**: A copy of the value is passed. Changes inside the method do not affect the original variable.
- **Object references**: The method receives a copy of the reference, which allows the method to modify the object's fields.

Example:

```
class Demo {
   void modifyPrimitive(int num) {
      num = num + 10;  // Only local change
   }

   void modifyObject(Sample obj) {
      obj.value = obj.value + 10;  // Changes the object's field
   }
}

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

      int num = 10;
      demo.modifyPrimitive(num);
      System.out.println("Primitive value after method call: " + num);  // Output: 10

      Sample obj = new Sample(20);
      demo.modifyObject(obj);
      System.out.println("Object value after method call: " + obj.value);  // Output: 30
   }
}
```

3.10 Using Objects as Parameters in Java

In Java, objects can be passed to methods as parameters. When an object is passed to a method, a **reference** to the object is passed, not the actual object itself. This means changes made to the object inside the method will affect the original object.

Example:

```
class Sample {
    int value;

    Sample(int value) {
        this.value = value;
    }

    void modify(Sample obj) {
        obj.value += 10;  // Modify the passed object's field
    }
}

public class Main {
    public static void main(String[] args) {
        Sample s = new Sample(20);
        System.out.println("Before: " + s.value);  // Output: 20

        s.modify(s);  // Passing object as parameter
        System.out.println("After: " + s.value);  // Output: 30
    }
}
```

3.11 Returning Objects

A method in Java can return an object. This is useful for creating and initializing objects within a method and returning them for use elsewhere.

Example:

```
class Sample {
    int value;

    Sample(int value) {
        this.value = value;
    }

    Sample createNewSample(int value) {
        return new Sample(value);  // Returning a new object
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Sample s1 = new Sample(10);

        Sample s2 = s1.createNewSample(50);  // Return a new object
        System.out.println("New object's value: " + s2.value);  // Output: 50
    }
}
```

---

4. Recursion

Recursion in Java occurs when a method calls itself either directly or indirectly to solve a problem. Recursion is commonly used to solve problems like calculating factorials, Fibonacci numbers, or traversing data structures like trees.

Key Components:

- **Base case**: The condition where recursion stops.
- **Recursive case**: The part where the method calls itself.

Example: Factorial Calculation

```
class Factorial {
    int calculateFactorial(int n) {
        if (n == 1) {
            return 1;  // Base case
        }
        return n * calculateFactorial(n - 1);  // Recursive call
    }
}

public class Main {
    public static void main(String[] args) {
        Factorial f = new Factorial();
        int result = f.calculateFactorial(5);  // 5 * 4 * 3 * 2 * 1 = 120
        System.out.println("Factorial of 5: " + result);  // Output: 120
    }
}
```

Caution:

- Ensure recursion has a base case to prevent infinite recursion and StackOverflowError.
- Consider the stack depth and avoid deep recursion for large inputs.

Advantages of Recursion:

- Simplifies code for problems like tree traversal or factorial calculation.

Disadvantages of Recursion:

- Can lead to stack overflow if the base case is not reached.
- May be less efficient than iterative solutions due to overhead from function calls.

5. static keyword

The static keyword in Java is used to indicate that a member (field, method, or nested class) belongs to the class itself rather than to instances of the class. This means that:

- **Static Fields**: These are shared across all instances of the class. They are initialized only once and are stored in the memory allocated to the class.

  ```
  public class Example {
      static int counter = 0; // Shared among all objects
  }
  ```

- **Static Methods**: These can be called without creating an instance of the class. They cannot access instance variables directly (since they don't belong to an object).

  ```
  public class Example {
      static void greet() {
          System.out.println("Hello!");
      }
  }
  ```

- **Static Block**: Used to initialize static fields when the class is loaded.

  ```
  public class Example {
      static {
          System.out.println("Class loaded!");
      }
  }
  ```

- **Static Nested Classes**: A nested class declared static does not need a reference to an outer class instance. More on this below.

---

6. final keyword

The final keyword is used to restrict modification. It can be applied to variables, methods, and classes:

- **Final Variables**: Once initialized, their values cannot be changed (constants).

```
public class Example {
    final int CONSTANT = 10; // Can't change this value
}
```

- **Final Methods**: Cannot be overridden by subclasses.

```
public class Example {
    public final void show() {
        System.out.println("This cannot be overridden.");
    }
}
```

- **Final Classes**: Cannot be subclassed.

```
public final class Example {
    // This class cannot have subclasses.
}
```

---

7. Nested Classes

Nested classes are classes declared within another class. They are used for logical grouping of classes and to access private members of the enclosing class.

- **Static Nested Classes**: Behave like top-level classes but are scoped within the enclosing class. They can be accessed without an instance of the enclosing class.

```
public class Outer {
    static class StaticNested {
        void display() {
            System.out.println("Static Nested Class");
        }
    }
}
```

- **Inner Classes**: Associated with an instance of the enclosing class. They have access to its members, even private ones.

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

- **Anonymous Classes**: A class defined and instantiated in a single statement, usually for one-time use.

```
Runnable r = new Runnable() {
    @Override
    public void run() {
        System.out.println("Anonymous Class");
    }
};
```

---

4. String Class

The String class in Java represents a sequence of characters. Strings are immutable, meaning their content cannot be changed once created.

- **Creating Strings**:

```
String s1 = "Hello"; // String literal
String s2 = new String("World"); // Using constructor
```

- **String Methods**:
  - length(): Returns the number of characters.
  - charAt(int index): Retrieves the character at a given index.
  - substring(int start, int end): Extracts a portion of the string.
  - toUpperCase(), toLowerCase(): Changes case.

```
String s = "Java";
System.out.println(s.length()); // 4
System.out.println(s.toUpperCase()); // JAVA
```

- **String Pool**: Java optimizes memory by maintaining a "pool" of string literals. If a string literal already exists in the pool, it reuses the same reference.

---

5. Command-Line Arguments

Command-line arguments are inputs passed to the main() method when running a Java program. They are passed as an array of String objects (String[] args).

- **Example**:

```
public class CommandLineExample {
    public static void main(String[] args) {
        for (String arg : args) {
            System.out.println(arg);
```

```
        }
    }
}
```

If you run this program with:

bash
CommandLineExample Hello World

Output:

Hello
World

- They are useful for providing configuration or data to programs without requiring user interaction during runtime.

Summary: This chapter provides the building blocks for object-oriented design in Java. Next, we will dive deeper into concepts like inheritance and polymorphism.

3.9 Unit Summary

Classes and Objects

- **Classes:** Blueprints for creating objects; they define properties (fields) and behaviors (methods).
  Example:

```
class Car {
    String model;
    void drive() {
        System.out.println("Driving the car");
    }
}
```

- **Objects:** Instances of classes created using the new keyword. Example: Car myCar = new Car();

Constructor, Overloading Constructor

- **Constructor:** A special method in a class that is automatically called when an object is created. It initializes object properties.

Example:

```
class Car {
   String model;
   Car(String m) {
      model = m;
   }
}
```

- **Overloading Constructor:** Multiple constructors in a class with different parameter lists to handle various initialization scenarios.

  SExample:

```
class Car {
   Car() {}
   Car(String model) {}
}
```

Using Object as Parameters, Argument Passing, Returning Objects

- **Using Objects as Parameters:** Pass objects to methods to operate on their data. Example:

```
void displayCar(Car c) {
   System.out.println(c.model);
}
```

- **Argument Passing:** Objects are passed by reference, meaning changes made within a method affect the original object.
- **Returning Objects:** Methods can return objects. Example:

```
Car getCar() {
   return new Car("Model X");
}
```

Recursion

- **Definition:** A process where a method calls itself. Useful for solving problems like factorial calculation, Fibonacci series, etc.

  Example:

```
int factorial(int n) {
   if (n == 1) return 1;
   return n * factorial(n - 1);  }
```

Introducing Access Control (public, private, and protected)

- **Public:** Members accessible from anywhere.
- **Private:** Members accessible only within the class.
- **Protected:** Members accessible within the package and subclasses.

Static, Final, Nested Classes, String Class, Command-line Arguments

- **Static:** A modifier indicating that a member belongs to the class rather than an instance.
- **Final:** Prevents inheritance for classes, overrides for methods, and reassignment for variables.
- **Nested Classes:** Classes defined inside other classes. They can be static or non-static.
- **String Class:** Represents immutable character sequences. Example: String s = "Hello";
- **Command-line Arguments:** Parameters passed to the main method when executing a program.
  Example:

```
public static void main(String[] args) {
   System.out.println(args[0]);
}
```

Command-line Arguments, The this Keyword, Garbage Collection

- **Command-line Arguments:** Enable the user to provide inputs while running the program from the terminal.
- **this Keyword:** Refers to the current instance of the class, resolving conflicts between instance variables and method parameters.
  Example:

```
class Car {
   String model;
   Car(String model) {
      this.model = model;
   }
}
```

- **Garbage Collection:** An automatic process that reclaims memory by removing objects no longer in use.

3.10 Check your progress

Multiple Choice Questions (MCQs)

1. **What is a class in object-oriented programming?**
   a) A method of organizing data

b) A blueprint or prototype for creating objects
c) A function that performs an operation
d) None of the above
Answer: b

2. **What is a constructor in Java?**
   a) A method used to destroy objects
   b) A special method to initialize an object
   c) A method for accessing private members
   d) None of the above
   Answer: b
3. **Which of the following is true about method overloading?**
   a) Methods must have the same number of parameters
   b) Method names must differ but parameter types should be the same
   c) Methods with the same name but different parameter lists
   d) None of the above
   Answer: c
4. **What is recursion in programming?**
   a) A technique for avoiding loops
   b) A method that calls itself
   c) A way to overload constructors
   d) None of the above
   Answer: b
5. **What is the purpose of the static keyword in Java?**
   a) To indicate a method belongs to a specific object instance
   b) To indicate a variable or method belongs to the class rather than any instance
   c) To prevent access to variables
   d) None of the above
   Answer: b
6. **Which access modifier provides the most restricted access?**
   a) public
   b) private
   c) protected
   d) default
   Answer: b
7. **What is garbage collection in Java?**
   a) Collecting unused objects to free up memory
   b) Manually deleting objects
   c) A method for cleaning input data
   d) None of the above
   Answer: a
8. **What does the final keyword do when used with a class?**
   a) Prevents the class from being instantiated
   b) Prevents the class from being inherited
   c) Prevents the class from accessing private members
   d) None of the above
   Answer: b

9. **Which keyword is used to reference the current object in a class?**
   a) this
   b) self
   c) super
   d) current
   Answer: a
10. **How are command-line arguments passed to a Java program?**
    a) Using args as parameters in the main() method
    b) Using a Scanner object
    c) By calling a special constructor
    d) None of the above
    Answer: a

True/False Questions

1. A constructor can be overloaded.
   Answer: True
2. Access modifiers control the visibility and accessibility of a class or class members.
   Answer: True
3. The static keyword can only be applied to methods, not variables.
   Answer: False
4. A final variable cannot be modified once initialized.
   Answer: True
5. Recursion is a technique where a function calls another function.
   Answer: False (It calls itself.)

Fill in the Blanks

1. A/an _____ is a blueprint for creating objects.
   Answer: class
2. The method used for initializing an object is called a _____.
   Answer: constructor
3. The _____ keyword is used to indicate that a variable or method belongs to the class rather than any instance.
   Answer: static
4. The _____ keyword in Java refers to the current object of the class.
   Answer: this
5. _____ collection automatically reclaims unused objects to free up memory.
   Answer: Garbage

Unit 4: **Inheritance**

4.1 Introduction
4.2 Member access and inheritance
4.3 Method overriding, dynamic method dispatch
4.4 Using abstract classes, using final with inheritance
4.5 The Object class; Packages, Interface, classpath.
4.6 Unit Summary
4.7 Check your progress

Inheritance and Related Concepts in Object-Oriented Programming

Inheritance is one of the cornerstones of Object-Oriented Programming (OOP). It allows a class (child or subclass) to acquire the properties and behaviors of another class (parent or superclass). This chapter explores critical inheritance topics such as member access, method overriding, dynamic method dispatch, abstract classes, and the use of the final keyword. Additionally, it delves into Java-specific features like the Object class, packages, interfaces, and classpaths.

---

1. Member Access and Inheritance

Access Modifiers

In Java, access to class members (fields, methods, and constructors) is controlled by **access modifiers**:

- **Private**: Accessible only within the class where it is defined.
- **Default (Package-private)**: Accessible within the same package.
- **Protected**: Accessible within the same package and by subclasses.
- **Public**: Accessible from any other class.

When a subclass inherits from a parent class, the following rules apply:

1. **Private members** are not directly accessible but can be accessed via public or protected methods in the parent class.
2. **Protected members** are accessible to the subclass, even if they are in different packages.
3. **Public members** are always accessible.

Example:

```
class Parent {
    private String privateField = "Private";
    protected String protectedField = "Protected";
    public String publicField = "Public";

    protected String getPrivateField() {
        return privateField;
```

```
    }
}

class Child extends Parent {
    void displayFields() {
        // System.out.println(privateField); // Error: privateField is not accessible.
        System.out.println(getPrivateField()); // Access private through a protected method.
        System.out.println(protectedField); // Accessible due to protected access.
        System.out.println(publicField); // Accessible due to public access.
    }
}
```

---

2. Method Overriding

Method overriding occurs when a subclass provides a specific implementation for a method already defined in its superclass. The overridden method must have:

- The same name, parameter list, and return type (or covariant return type).
- The same or broader access level.
- The @Override annotation (optional but recommended).

Example:

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

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

Key Points:

- **Superclass Method Call**: Use super.methodName() to call the superclass method.
- **Polymorphism**: Overridden methods support runtime polymorphism, enabling dynamic behavior based on object type.

---

3. Dynamic Method Dispatch

Dynamic method dispatch (also known as **runtime polymorphism**) refers to the process where the method to be executed is determined at runtime based on the actual object type, not the reference type.

Example:

```
class Parent {
    void show() {
        System.out.println("Parent show");
    }
}

class Child extends Parent {
    @Override
    void show() {
        System.out.println("Child show");
    }
}

public class Main {
    public static void main(String[] args) {
        Parent obj = new Child(); // Reference type: Parent, Object type: Child
        obj.show(); // Output: "Child show"
    }
}
```

---

4. Using Abstract Classes

An **abstract class** is a class that cannot be instantiated and may contain abstract methods (methods without a body). Subclasses must provide implementations for these methods unless they are also abstract.

Example:

```
abstract class Shape {
    abstract void draw(); // Abstract method
}

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

Key Features:

- Abstract classes can have both abstract and concrete methods.
- They can define common functionality for subclasses while enforcing specific behaviors through abstract methods.

---

5. Using final with Inheritance

The final keyword restricts inheritance and modification:

1. **Final Class**: A final class cannot be extended.

   ```
   final class Constants { }
   // class ExtendedConstants extends Constants {} // Error
   ```

2. **Final Method**: A final method cannot be overridden.

   ```
   class Parent {
      final void display() { }
   }
   class Child extends Parent {
      // void display() { } // Error
   }
   ```

3. **Final Variables**: Their values cannot be reassigned once initialized.

---

6. The Object Class

The Object class is the root class of the Java class hierarchy. Every class implicitly extends Object. Key methods include:

- **toString()**: Returns a string representation of the object.
- **equals(Object obj)**: Compares two objects for equality.
- **hashCode()**: Returns an integer hash code for the object.

---

7. Packages

Packages group related classes and interfaces to organize code and control access. Common package keywords:

- **package**: Defines the package a class belongs to.
- **import**: Imports classes from other packages.

Example:
```
package com.example.shapes;
public class Circle { }
```

To use the Circle class:

```
import com.example.shapes.Circle;
```

---

8. Interfaces

An **interface** is a reference type in Java that can contain constants, method signatures, and default methods. A class implements an interface using the implements keyword.

Example:
```
interface Vehicle {
   void start();
}

class Car implements Vehicle {
   @Override
   public void start() {
      System.out.println("Car starts");
   }
}
```

Key Points:

- A class can implement multiple interfaces.
- Interfaces support multiple inheritance, unlike classes.

---

9. Classpath

The **classpath** defines where the Java compiler and runtime look for classes and packages. It can be specified:

- As an environment variable.
- Using the -cp or -classpath option when running a program.

Example:
```
java -cp /path/to/classes MainClass
```

Conclusion

Inheritance and its associated concepts—method overriding, dynamic dispatch, and the use of abstract classes—enable code reuse, polymorphism, and extensibility in OOP. When combined

with interfaces and packages, these tools provide a robust framework for building modular and maintainable applications. Understanding these principles is vital for mastering Java and other object-oriented languages.

Unit 5: Exception Handling in Java

Exception handling is a critical concept in Java that ensures the program can handle errors gracefully and continue execution without abrupt termination. This unit explores the fundamentals of exception handling, the types of exceptions, Java's built-in exceptions, and how to define custom exceptions.

---

1. Fundamentals of Exception Handling

- **Definition**: An exception is an abnormal condition that occurs during the execution of a program, disrupting the normal flow of instructions.
- **Why Exception Handling?**
  - Ensures program stability.
  - Provides mechanisms to handle errors at runtime.
  - Enables the separation of error-handling code from the main logic.
- **Key Concepts**:
  - **try**: Defines a block of code to monitor for exceptions.
  - **catch**: Handles exceptions that occur in the corresponding try block.
  - **finally**: Executes a block of code, regardless of whether an exception occurs.
  - **throw**: Used to explicitly throw an exception.
  - **throws**: Declares exceptions a method might throw.
- **Basic Syntax**:

```
try {
    // Code that might throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
} finally {
    // Code that will always execute
}
```

---

2. Types of Exceptions

Exceptions in Java can be broadly categorized into two types:

1. **Checked Exceptions**:
   - These are exceptions that the compiler checks at compile time.
   - Must be handled using try-catch or declared using throws.
   - Examples: IOException, SQLException.
2. **Unchecked Exceptions**:
   - These exceptions occur at runtime and are not checked at compile time.
   - Subclasses of RuntimeException.
   - Examples: NullPointerException, ArithmeticException.
3. **Errors**:
   - Represent serious issues that cannot typically be handled by the application.

- Examples: OutOfMemoryError, StackOverflowError.

---

3. Java's Built-in Exceptions

Java provides a rich hierarchy of built-in exceptions, which can be broadly classified into:

- **Arithmetic Exceptions**: e.g., ArithmeticException (divide by zero).
- **Input/Output Exceptions**: e.g., IOException, FileNotFoundException.
- **Null Pointer Exceptions**: e.g., NullPointerException (accessing an object reference that is null).
- **Array Exceptions**: e.g., ArrayIndexOutOfBoundsException.
- **Class Exceptions**: e.g., ClassNotFoundException.

Example:

```
public class BuiltInExceptionExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // Throws ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

---

4. User-Defined Exceptions

Java allows developers to create their own exceptions by extending the Exception class (for checked exceptions) or RuntimeException class (for unchecked exceptions).

- **Steps to Create a Custom Exception**:
    1. Define a class that extends Exception or RuntimeException.
    2. Provide constructors for the exception class.
    3. Use throw to trigger the exception where necessary.

Example:

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

public class UserDefinedExceptionExample {
    public static void main(String[] args) {
        try {
            validateAge(15);
        } catch (CustomException e) {
            System.out.println("Caught Exception: " + e.getMessage());
        }
    }

    static void validateAge(int age) throws CustomException {
        if (age < 18) {
            throw new CustomException("Age must be at least 18.");
```

```
        }
    }
}
```

4.6 Unit summary

Exception handling in Java is a powerful mechanism that improves the robustness and reliability of applications. By understanding the fundamentals, built-in exceptions, and creating user-defined exceptions, developers can ensure better error management and a smoother user experience.

4.7 Check your progress:

Multiple Choice Questions (MCQs)

1. **Which of the following access modifiers allows access to members within the same package and by subclasses in other packages?**
   a) Private
   b) Protected
   c) Public
   d) Default

   Answer: b) Protected

2. **What is required for a method in a subclass to override a method in its superclass?**
   a) Same name and parameter list
   b) Same return type or covariant return type
   c) Same or broader access level
   d) All of the above

   Answer: d) All of the above

3. **What does the @Override annotation in Java indicate?**
   a) The method is an abstract method.
   b) The method is overriding a superclass method.
   c) The method is final.
   d) The method is private.

   Answer: b) The method is overriding a superclass method.

4. **What is the output of the following code?**

   java
   Copy code
   Parent obj = new Child();
   obj.show();

   a) Parent show
   b) Child show
   c) Compilation error
   d) Runtime error

   Answer: b) Child show

5. **Which keyword in Java prevents a class from being extended?**
   a) Abstract

b) Final
c) Static
d) Protected

Answer: b) Final

6. **Which method in the Object class is used to compare two objects for equality?**
   a) hashCode()
   b) toString()
   c) equals()
   d) compare()

   Answer: c) equals()

7. **Which Java feature allows multiple classes to implement the same behavior?**
   a) Abstract classes
   b) Interfaces
   c) Inheritance
   d) Packages

   Answer: b) Interfaces

8. **What is the classpath used for?**
   a) To specify the path of the source code files
   b) To define where the Java compiler and runtime locate classes and packages
   c) To identify the main method in a Java program
   d) None of the above

   Answer: b) To define where the Java compiler and runtime locate classes and packages

9. **What is the role of the finally block in exception handling?**
   a) It executes only if an exception occurs.
   b) It executes only if no exception occurs.
   c) It always executes, regardless of exceptions.
   d) It declares the exceptions a method might throw.

   Answer: c) It always executes, regardless of exceptions.

10. **Which of the following is a checked exception in Java?**
    a) NullPointerException
    b) ArithmeticException
    c) IOException
    d) ArrayIndexOutOfBoundsException

    Answer: c) IOException

Fill in the Blanks

1. The process of defining a method in a subclass that has the same name and signature as in its superclass is called _____.
   Answer: Method Overriding
2. The _____ class is the root class of the Java class hierarchy.
   Answer: Object
3. _____ methods in abstract classes do not have a body and must be implemented by subclasses.

Answer: Abstract

4. The _____ keyword in Java is used to define constants or restrict inheritance.
   Answer: Final
5. _____ refers to the runtime decision of which method to invoke based on the object type.
   Answer: Dynamic Method Dispatch

True/False Questions

1. The protected access modifier allows access to members only within the same package.
   Answer: False
2. A class can implement multiple interfaces in Java. **Answer:** True
3. Abstract classes in Java cannot have concrete methods. **Answer:** False
4. The hashCode() method in the Object class returns a string representation of the object.
   Answer: False
5. The try block in exception handling must always be followed by a catch block.
   Answer: False (it can also be followed by a finally block or both).

Unit 5: Exception handling
5.1 Introduction
5.2 Exception handling Fundamentals
5.3 Exception types
5.4 Java's built-in exceptions
5.5 User defined exceptions.
5.6 Unit Summary
5.7 Check your progress

Unit 5: Java Exception Handling

5.1 Introduction

- Definition of exceptions
- Importance of exception handling in Java
- Common scenarios where exception handling is necessary
- Difference between errors and exceptions

Definition of Exceptions

In programming, an exception is an event that disrupts the normal flow of execution in a program. It occurs when a program encounters an error during runtime that it cannot handle on its own. In Java, exceptions are objects that represent these runtime errors, and they are used to signal and handle unexpected situations systematically.

Importance of Exception Handling in Java

Exception handling is a critical feature in Java that ensures the robustness and reliability of applications. It provides a structured mechanism to:

1. Detect and respond to runtime errors without crashing the program.
2. Separate error-handling code from the main program logic, improving readability and maintainability.
3. Allow programs to recover from unexpected conditions gracefully, ensuring a better user experience.
4. Facilitate debugging by providing detailed information about errors using stack traces.

By using try-catch blocks, developers can control the flow of execution when exceptions occur and ensure resources are released properly through finally blocks or using the try-with-resources statement.

Common Scenarios Where Exception Handling Is Necessary

1. **File Operations**: Handling errors when reading from or writing to files, such as file not found or insufficient permissions.
2. **Database Connectivity**: Managing issues like invalid SQL queries or connection timeouts.
3. **Invalid User Input**: Handling incorrect or unexpected user-provided data.
4. **Network Operations**: Dealing with connectivity issues, timeouts, or unreachable servers during API calls.
5. **Arithmetic Operations**: Managing errors such as division by zero.
6. **Array and Collections**: Handling index-out-of-bounds or concurrent modification exceptions.
7. **Resource Management**: Ensuring proper release of system resources like file streams or database connections, even in case of errors.

Difference Between Errors and Exceptions

Aspect	Error	Exception
Definition	Serious problems that arise during program execution and are often beyond the control of the application.	Recoverable issues that occur during runtime, which the program can handle.
Examples	OutOfMemoryError, StackOverflowError	NullPointerException, IOException, SQLException
Recovery	Errors are generally not recoverable by the application.	Exceptions can often be caught and handled programmatically.
Hierarchy	Part of the java.lang.Error class and its subclasses.	Part of the java.lang.Exception class and its subclasses.
Causes	Related to system-level issues like resource exhaustion.	Related to application logic or user errors.

5.2 Exception Handling Fundamentals

- Overview of Java's exception-handling mechanism
- Key concepts:
  - try, catch, finally blocks
  - throw vs. throws
- Flow of control in exception handling
- Best practices for exception handling

Overview of Java's Exception-Handling Mechanism

Java's exception-handling mechanism provides a structured way to manage runtime errors, ensuring the smooth functioning of programs. It enables developers to detect, handle, and recover from unforeseen conditions without crashing the program. The mechanism revolves around the **try-catch-finally** structure and allows developers to propagate exceptions using the throw and throws keywords.

Key Concepts

1. *try-catch-finally Blocks*

1. try, catch, finally Blocks

- **try Block**:
  Contains the code that might throw an exception. If an exception occurs within the try block, control transfers to the corresponding catch block.

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

- **catch Block**:
  Handles the exception. Each catch block is associated with a specific exception type or hierarchy.

  ```
  catch (ArithmeticException e) {
      System.out.println("Cannot divide by zero.");
  }
  ```

- **finally Block**:
  Executes regardless of whether an exception occurs. It is typically used for cleanup actions like closing files or releasing resources.

  ```
  finally {
      System.out.println("Execution complete.");
  }
  ```

2. throw vs. throws

- **throw**: Used to explicitly throw an exception from a method or block of code.

  ```
  throw new IllegalArgumentException("Invalid argument");
  ```

- **throws**: Declares exceptions that a method might throw, informing the caller to handle or propagate them further.

  ```
  public void readFile() throws IOException {
      // File reading code
  }
  ```

Flow of Control in Exception Handling

1. Code in the try block is executed.
2. If an exception occurs:
   - Control jumps to the first catch block matching the exception type.
   - If no matching catch block exists, the exception propagates up the call stack.
3. The finally block executes after the try and catch blocks, regardless of whether an exception was thrown or handled.
4. If no exception occurs, the catch block is skipped, and execution moves to the finally block.

Example:

```
try {
    int[] numbers = {1, 2, 3};
    System.out.println(numbers[5]); // ArrayIndexOutOfBoundsException
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index is out of bounds.");
```

```
} finally {
    System.out.println("End of try-catch block.");
}
```

Best Practices for Exception Handling

1. **Use Specific Exceptions**: Catch specific exceptions rather than a generic Exception to improve clarity.
2. **Avoid Silent Failures**: Avoid empty catch blocks; always log or handle exceptions appropriately.

   ```
   java
   Copy code
   catch (IOException e) {
       e.printStackTrace(); // Log the exception
   }
   ```

3. **Minimize the Scope of try Blocks**: Keep the try block as small as possible to isolate error-prone code.
4. **Don't Swallow Exceptions**: Avoid ignoring exceptions; this can make debugging difficult.
5. **Use Finally for Resource Management**: Always close resources in the finally block or use try-with-resources (introduced in Java 7).

```
try (FileReader reader = new FileReader("file.txt")) {
    // File processing
} catch (IOException e) {
    e.printStackTrace();
}
```

6. **Avoid Overusing Exceptions**: Use exceptions for exceptional conditions, not as a substitute for logic.
7. **Document Exceptions**: Clearly document the exceptions your method might throw using Javadoc comments.

5.3 Exception Types

- Classification of exceptions:
  - Checked exceptions (e.g., IOException, SQLException)
  - Unchecked exceptions (e.g., ArithmeticException, NullPointerException)
- Errors in Java (e.g., OutOfMemoryError, StackOverflowError)
- Differences between checked, unchecked exceptions, and errors

Classification of Exceptions in Java

In Java, exceptions are categorized based on whether they need to be handled by the programmer or not. These categories include **Checked Exceptions**, **Unchecked Exceptions**, and **Errors**.

---

1. Checked Exceptions

Checked exceptions are exceptions that must be either caught or declared in the method signature using the throws keyword. These exceptions are typically situations that are beyond the program's control but are anticipated and can be handled appropriately. If a method can potentially throw a checked exception, the calling code must handle or declare it explicitly.

- **Examples:**
  - **IOException**: Occurs when an input-output operation fails, such as reading from a file or network.
  - **SQLException**: Occurs when there is a database access error, such as failure in executing SQL queries.

  Example code:

  ```
  import java.io.*;

  public class CheckedExceptionExample {
  ```

```
    public void readFile() throws IOException {
        FileReader file = new FileReader("example.txt");
        BufferedReader fileInput = new BufferedReader(file);
        System.out.println(fileInput.readLine());
        fileInput.close();
    }

    public static void main(String[] args) {
        try {
            CheckedExceptionExample obj = new CheckedExceptionExample();
            obj.readFile();
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }
}
```

---

2. Unchecked Exceptions

Unchecked exceptions, also called **runtime exceptions**, do not need to be explicitly caught or declared. These exceptions are typically due to programming errors, such as logical errors or invalid operations that can be avoided during normal program execution. The compiler does not force the developer to handle them.

- **Examples:**
    - **ArithmeticException**: Occurs when there is an arithmetic error, such as dividing by zero.
    - **NullPointerException**: Occurs when an application attempts to use a null object reference, such as calling a method on null.

Example code:

```
public class UncheckedExceptionExample {
    public void divideNumbers() {
        int result = 10 / 0;  // ArithmeticException
    }

    public static void main(String[] args) {
        try {
            UncheckedExceptionExample obj = new UncheckedExceptionExample();
            obj.divideNumbers();
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

---

3. Errors in Java

Errors in Java represent serious problems that a program generally cannot recover from. Errors are typically related to system-level issues that are out of the application's control, such as running out of memory or a stack overflow.

- **Examples:**
  - **OutOfMemoryError**: This occurs when the Java Virtual Machine (JVM) runs out of memory.
  - **StackOverflowError**: This occurs when a stack overflow happens, often due to deep or infinite recursion.

Example of StackOverflowError:

```
public class StackOverflowExample {
    public void recurse() {
        recurse();  // This will cause a StackOverflowError
    }

    public static void main(String[] args) {
        StackOverflowExample obj = new StackOverflowExample();
        obj.recurse();
    }
}
```

Differences Between Checked Exceptions, Unchecked Exceptions, and Errors

Feature	Checked Exceptions	Unchecked Exceptions	Errors
Definition	Exceptions that must be explicitly handled or declared	Exceptions that are not required to be explicitly handled	Serious problems that cannot be typically recovered from
Inheritance	Extends Exception but not RuntimeException	Extends RuntimeException	Extends Error
Handling Requirement	Must be either caught with a try-catch block or declared with throws in method signature	No requirement to be caught or declared	No requirement to handle or declare
Examples	IOException, SQLException	NullPointerException, ArrayIndexOutOfBoundsException	OutOfMemoryError, StackOverflowError

Feature	Checked Exceptions	Unchecked Exceptions	Errors
Common Causes	External conditions like I/O failure or database issues	Programming errors such as dividing by zero or accessing null	System-level issues like insufficient memory or stack limits
Can They Be Avoided?	Yes, the program can handle or avoid them by proper coding	Mostly avoidable through proper code logic	Typically cannot be avoided by the application

---

Summary

- **Checked Exceptions** need to be explicitly handled or declared, typically due to external conditions like I/O or database issues.
- **Unchecked Exceptions** are usually logical errors that can be avoided with better coding practices and do not require mandatory handling.
- **Errors** are catastrophic failures, often related to system resources or the environment, and usually cannot be handled by the program.

Understanding these distinctions helps in writing robust Java applications by ensuring that exceptions are appropriately handled, and that serious system-level failures are properly managed.

5.4 Java's Built-in Exceptions

- Commonly used built-in exceptions:
  - ArithmeticException
  - NullPointerException
  - ArrayIndexOutOfBoundsException
  - ClassNotFoundException
  - IOException
  - FileNotFoundException
- Practical examples and use cases
- How to interpret exception stack traces

Java's Built-in Exceptions

Java provides a wide range of built-in exceptions, which are pre-defined classes that handle specific types of errors in the program. These exceptions are part of the Java standard library and are designed to help manage runtime errors. Below, we'll cover some of the commonly used built-in exceptions, practical examples, and how to interpret exception stack traces.

Commonly Used Built-in Exceptions

1. ArithmeticException

- **Description**: Thrown when an exceptional arithmetic condition occurs, such as division by zero.
- **Example**:

```
public class ArithmeticExample {
    public static void main(String[] args) {
        int a = 10;
        int b = 0;
        System.out.println(a / b);  // This will throw ArithmeticException
    }
}
```

- **Use case**: Occurs when you attempt to divide a number by zero, which is mathematically undefined.

2. NullPointerException

- **Description**: Thrown when the JVM attempts to access or modify an object reference that is null.
- **Example**:

```
public class NullPointerExample {
    public static void main(String[] args) {
        String str = null;
        System.out.println(str.length());  // This will throw NullPointerException
    }
}
```

- **Use case**: Happens when you attempt to call a method or access a field on a null object reference.

3. ArrayIndexOutOfBoundsException

- **Description**: Thrown when an array is accessed with an illegal index (negative or greater than the array's size).
- **Example**:

```
public class ArrayIndexExample {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3};
        System.out.println(arr[5]);  // This will throw ArrayIndexOutOfBoundsException
```

```
    }
}
```

- **Use case**: Occurs when you try to access an index that is out of the bounds of the array.

4. ClassNotFoundException

- **Description**: Thrown when an application tries to load a class by its name, but the class cannot be found.
- **Example**:

```
public class ClassNotFoundExample {
    public static void main(String[] args) {
        try {
            Class.forName("com.example.NonExistentClass");        // This will throw ClassNotFoundException
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

- **Use case**: Typically thrown when attempting to dynamically load a class that is not available in the classpath.

5. IOException

- **Description**: Thrown when an I/O (Input/Output) operation fails or is interrupted.
- **Example**:

```
import java.io.*;

public class IOExceptionExample {
    public static void main(String[] args) {
        try {
            FileInputStream file = new FileInputStream("nonexistentfile.txt");  // This will throw IOException
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **Use case**: Happens during file or stream operations, such as when a file cannot be read or written to.

6. FileNotFoundException

- **Description**: A subclass of IOException, it is thrown when a file with the specified pathname does not exist.
- **Example**:

```
import java.io.*;

public class FileNotFoundExample {
   public static void main(String[] args) {
      try {
         FileReader fr = new FileReader("file_that_does_not_exist.txt");  // This will throw FileNotFoundException
      } catch (FileNotFoundException e) {
         e.printStackTrace();
      }
   }
}
```

- **Use case**: Commonly occurs when trying to access a file that doesn't exist in the given path.

Practical Examples and Use Cases

- **ArithmeticException**: When building a calculator, division by zero should be handled properly to prevent the application from crashing. By catching this exception, we can display a meaningful error message or handle the issue in an alternate way.
- **NullPointerException**: It often occurs when developers forget to check if an object is null before using it. A good practice is to validate objects before calling methods or accessing fields.
- **ArrayIndexOutOfBoundsException**: This exception can happen in programs that use arrays or lists. It's common when the user enters invalid input or when iterating beyond the array bounds. You can prevent this by checking the index before accessing an array.
- **ClassNotFoundException**: When working with reflection or dynamic class loading, it's essential to handle this exception to ensure the application runs smoothly if the class is not found.
- **IOException** and **FileNotFoundException**: When reading from or writing to a file, handling these exceptions is crucial for file handling tasks like logging, configuration file reading, or writing data to external storage.

How to Interpret Exception Stack Traces

A **stack trace** is a report that is generated when an exception is thrown. It shows the sequence of method calls that led to the exception. Here's an example of a typical stack trace:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ArithmeticExample.main(ArithmeticExample.java:6)
```

- **Exception Type**: java.lang.ArithmeticException is the type of exception thrown.
- **Description**: / by zero indicates the cause of the exception (i.e., division by zero).
- **Stack Trace**:
    - The trace shows the sequence of method calls.
    - at ArithmeticExample.main(ArithmeticExample.java:6) tells you that the exception occurred at line 6 in the main method of the ArithmeticExample class.

How to Read the Stack Trace:

1. **Exception Type**: The first line identifies the exception type and provides a brief description of the error.
2. **Method Calls**: Each subsequent line represents a method call in the call stack, starting from the most recent (where the exception occurred) to the initial method call.
3. **File Name and Line Number**: The part (ArithmeticExample.java:6) shows the file name and line number where the error occurred, helping developers quickly locate the issue in their code.

Conclusion

Java's built-in exceptions are an essential tool for handling errors and ensuring that programs run reliably. By understanding common exceptions like ArithmeticException, NullPointerException, and others, developers can prevent crashes and handle runtime issues more gracefully. Interpreting stack traces is equally important for troubleshooting, as it allows developers to pinpoint where the error originated in their code and fix it promptly.

5.5 User-Defined Exceptions

- Purpose and scenarios for creating custom exceptions
- Steps to define a user-defined exception:
    1. Extend the Exception or RuntimeException class
    2. Create a constructor for custom messages
- Example: Creating a CustomException class
- Handling and throwing user-defined exceptions in code

User-Defined Exceptions

User-defined exceptions in Java (and other object-oriented languages) allow developers to create their own exception classes tailored to the specific needs of their applications. These exceptions can be used to handle errors or exceptional situations that are not covered by standard Java exceptions like NullPointerException, IOException, etc.

Purpose and Scenarios for Creating Custom Exceptions

Creating user-defined exceptions provides several advantages:

1. **Specific Error Reporting**: It allows for more meaningful and specific error messages, making it easier to debug and handle exceptional cases in the code.
2. **Custom Handling**: They can represent business logic exceptions or errors specific to the application domain, like "InsufficientFundsException" or "InvalidOrderStateException".
3. **Better Control over Exceptions**: Allows developers to have full control over how exceptions are thrown and caught, improving the robustness and clarity of error-handling.

Some scenarios where custom exceptions may be useful:

- **Validating business rules**: For example, throwing a custom exception when a user tries to withdraw more money than available in their account.
- **Invalid input**: When input fails to meet specific criteria, a custom exception can be thrown to handle the error more clearly.
- **Database issues**: When handling database-specific errors that aren't captured by standard exceptions.

Steps to Define a User-Defined Exception

1. Extend the Exception or RuntimeException Class

- **Exception**: If the exception is checked (meaning it must be declared or caught), you should extend the Exception class.
- **RuntimeException**: If the exception is unchecked (not required to be declared or caught), extend the RuntimeException class.

2. Create a Constructor for Custom Messages

When creating a custom exception, you can define constructors to accept custom error messages or other information relevant to the exception, which can then be used in exception handling.

Example: Creating a CustomException Class

```
// Custom exception class that extends Exception
public class CustomException extends Exception {

    // Constructor with custom message
```

```
    public CustomException(String message) {
        super(message);  // Passing the message to the superclass constructor
    }
}
```

In the example above:

- CustomException extends Exception, meaning it is a checked exception.
- The constructor accepts a message that will be passed to the parent Exception class, which can later be used to retrieve the message when the exception is caught.

Handling and Throwing User-Defined Exceptions in Code

To use the custom exception, you can throw it in your code using the throw keyword, and then catch it with a try-catch block.

Throwing the Exception:

```
public class Account {
    private double balance;

    public Account(double balance) {
        this.balance = balance;
    }

    public void withdraw(double amount) throws CustomException {
        if (amount > balance) {
            // Throw custom exception if withdrawal exceeds balance
            throw new CustomException("Insufficient funds for withdrawal.");
        } else {
            balance -= amount;
            System.out.println("Withdrawal successful! New balance: " + balance);
        }
    }
}
```

Handling the Exception:

```
public class TestAccount {
    public static void main(String[] args) {
        Account myAccount = new Account(100.0);

        try {
            myAccount.withdraw(150.0);  // This will throw a CustomException
        } catch (CustomException e) {
            // Catch and handle the custom exception
            System.out.println("Error: " + e.getMessage());
        }
    }
```

}

Explanation:

- In the Account class, the withdraw method throws a CustomException if the withdrawal amount exceeds the available balance.
- The TestAccount class demonstrates how to handle the custom exception using a try-catch block.
- If the CustomException is thrown, the message "Insufficient funds for withdrawal." is caught and printed.

Key Points:

1. **Throwing a Custom Exception**: Use throw new CustomException("message") to create and throw an exception.
2. **Catching a Custom Exception**: Use try-catch blocks to catch the exception and handle it appropriately.
3. **Custom Messages**: You can pass a message to the exception constructor for detailed error information.

Conclusion

User-defined exceptions are a powerful tool in error handling, allowing you to manage specific business logic errors in a more organized and descriptive way. By extending either the Exception or RuntimeException class, you can create exceptions that provide more context, making debugging easier and ensuring your application handles exceptions in a meaningful manner.

Unit Summary

1. Importance of Exception Handling

Exception handling is a crucial aspect of writing robust and reliable code. It allows developers to anticipate and respond to errors gracefully, ensuring that the application continues to run smoothly even when unexpected conditions arise. Without exception handling, programs may crash, leading to a poor user experience and system failures. Proper exception handling provides:

- **Graceful error recovery:** Helps handle unforeseen problems without stopping the program.
- **Debugging aid:** Makes it easier to trace the root cause of errors.
- **Improved user experience:** Displays friendly error messages instead of generic crashes.

2. Exception Types and Their Relevance

Exceptions can be broadly categorized into:

- **Syntactical Errors:** These are errors detected by the compiler before the program is run, such as incorrect syntax or typos.
- **Runtime Errors:** These occur while the program is running (e.g., division by zero, null pointer dereferencing).
- **Logical Errors:** These occur when the program runs without crashing but produces incorrect results due to flawed logic.

Within runtime errors, exceptions can be further divided into:

- **Checked Exceptions:** These must be explicitly caught or declared in the method signature. They are often used for external issues like file handling or network problems (e.g., IOException in Java).
- **Unchecked Exceptions:** These are runtime exceptions, usually indicating programming bugs, such as NullPointerException or ArrayIndexOutOfBoundsException. They are not mandatory to handle but should be addressed to avoid unexpected crashes.

3. Built-in Exceptions vs. Custom Exceptions

- **Built-in Exceptions**: Programming languages typically provide a set of built-in exceptions for common error conditions. For example:
  - **Java**: NullPointerException, IOException, SQLException
  - **Python**: ValueError, IndexError, FileNotFoundError These exceptions provide predefined messages and behaviors that handle common error scenarios.
- **Custom Exceptions**: Sometimes, a specific error scenario may not be covered by built-in exceptions. In such cases, developers can define their own custom exceptions. This provides flexibility to handle unique conditions and ensures more descriptive and context-relevant error handling. Custom exceptions are usually derived from standard exception classes in the language.

4. Key Coding Practices for Exception Management

Good exception handling practices help improve both the reliability and maintainability of code. Key practices include:

- **Catch Specific Exceptions**: Avoid catching generic Exception or Throwable types unless absolutely necessary. Catch specific exceptions to provide tailored handling for different scenarios.
- **Avoid Empty Catch Blocks**: Never leave a catch block empty. It's important to either handle the exception or log it for future investigation.
- **Use Finally for Cleanup**: When resources such as file handles or network connections are used, employ the finally block to ensure they are properly closed or released, even if an exception occurs.
- **Logging Exceptions**: Always log exceptions with relevant context (e.g., error messages, stack traces) so that issues can be tracked and resolved.

- **Don’t Overuse Exceptions**: Exceptions are expensive in terms of performance. Use them sparingly and for actual error scenarios. Don’t use them for regular control flow in the program.
- **Create Meaningful Custom Exceptions**: When creating custom exceptions, ensure they provide clear, informative messages and are used only for specific, well-defined error conditions.

By applying these best practices, developers can write more reliable, maintainable, and understandable code.

5.7 Check Your Progress

- **Short Answer Questions:**
    1. Why is it necessary to use exception handling in Java programs?
    2. How do you create a custom exception class? Provide a simple code example.
    3. What is the difference between throw and throws?
- **Coding Exercise:**
    - Write a Java program that demonstrates the use of try, catch, and finally blocks.
    - Create and handle a user-defined exception for invalid user input.

MCQs:

1. **What is the purpose of exception handling in Java?**
    - a) To handle errors during runtime
    - b) To handle compile-time errors
    - c) To handle exceptions during design time
    - d) To improve performance

    Answer: a) To handle errors during runtime

2. **Which of the following is a checked exception in Java?**
    - a) NullPointerException
    - b) ArithmeticException
    - c) IOException
    - d) ArrayIndexOutOfBoundsException

    Answer: c) IOException

3. **Which keyword is used to define a block of code that can throw an exception?**
    - a) throws
    - b) throw
    - c) try
    - d) catch

Answer: a) throws

4. **Which of the following is a subclass of the Exception class?**
   - a) Error
   - b) Throwable
   - c) RuntimeException
   - d) IOException

   Answer: c) RuntimeException

5. **Which method is used to print the stack trace of an exception in Java?**
   - a) printStack()
   - b) print()
   - c) getMessage()
   - d) printStackTrace()

   Answer: d) printStackTrace()

6. **What is the default behavior when an exception is thrown but not handled?**
   - a) Program terminates
   - b) Program continues normally
   - c) It causes an infinite loop
   - d) It prints an error message and exits

   Answer: a) Program terminates

7. **What is the superclass of all exceptions in Java?**
   - a) Exception
   - b) Throwable
   - c) Object
   - d) Error

   Answer: b) Throwable

8. **Which of the following exceptions is unchecked?**
   - a) ClassNotFoundException
   - b) SQLException
   - c) ArrayIndexOutOfBoundsException
   - d) IOException

   Answer: c) ArrayIndexOutOfBoundsException

9. **Which of these is used to handle multiple exceptions in a single catch block in Java 7 and later?**
   - a) Multiple catch blocks
   - b) Throws clause

- c) Catch with | (pipe) operator
- d) Try-finally block

Answer: c) Catch with | (pipe) operator

10. **Which of the following statements about user-defined exceptions is true?**
    - a) User-defined exceptions are always checked exceptions
    - b) User-defined exceptions must extend the Exception class
    - c) User-defined exceptions cannot be used with the throw statement
    - d) User-defined exceptions cannot be handled with a try-catch block

    Answer: b) User-defined exceptions must extend the Exception class

Fill in the Blanks

1. **In Java, the _____ keyword is used to throw an exception explicitly.**
2. **A _____ exception is one that is not checked at compile time but is checked at runtime.**
3. **The _____ block is used to test a block of code for exceptions.**
4. **The _____ block follows the try block to handle exceptions.**
5. **In Java, exceptions are objects that inherit from the _____ class.**

Answers: 1. Throw, 2. Unchecked, 3. try, 4. catch, 5. throwable

True & False

1. In Java, you must handle all exceptions, including those that are unchecked, using try-catch blocks. **Answer**: False
2. A user-defined exception in Java is created by extending the Exception class. **Answer**: True
3. The finally block in Java is executed only if an exception is thrown. **Answer**: False
4. Runtime exceptions are also known as unchecked exceptions. **Answer**: True
5. The try block can exist without a corresponding catch block in Java. **Answer**: True

Unit 6: **Multithreaded Programming**

6.1 Introduction

6.2 The Java thread model (thread priorities, synchronization & inter-thread communication)

6.3 Deadlock

6.4 Thread Group

6.5 Unit Summary

6.6 Check your progress

6.1 Introduction to Multithreaded Programming

Multithreaded programming allows multiple threads (independent units of execution) to run concurrently within a single program. This enables a program to perform several tasks at once, improving efficiency and responsiveness, especially on multi-core processors. Threads share the same memory space, but each thread has its own execution path.

Key Concepts:

- **Thread**: A lightweight process that shares resources with other threads.
- **Concurrency**: Multiple threads running in overlapping time periods.
- **Parallelism**: Running threads simultaneously on multiple processors.

Example: Simple Thread

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

   public static void main(String[] args) {
      MyThread t1 = new MyThread();
      t1.start();  // Start the thread
   }
}
```

In this example, the thread MyThread is created and started. The run method contains the code that will execute when the thread starts.

6.2 The Java Thread Model

The Java thread model allows fine control over thread behavior, including priorities, synchronization, and inter-thread communication.

Thread Priorities:

Java threads have a priority, which is an integer value that affects the thread's scheduling. The default priority is Thread.NORM_PRIORITY (5), but you can adjust it between Thread.MIN_PRIORITY (1) and Thread.MAX_PRIORITY (10).

Example: Setting Thread Priority

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

    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.setPriority(Thread.MAX_PRIORITY);  // Set maximum priority
        t1.start();
    }
}
```

Synchronization:

Synchronization ensures that only one thread can access a resource at a time. This is essential to avoid data corruption when multiple threads modify shared data.

Example: Synchronizing a Method

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

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

    public synchronized int getCount() {
        return count;
    }
}

class MyThread extends Thread {
    Counter counter;

    public MyThread(Counter counter) {
        this.counter = counter;
    }

    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
```

```
        }
    }

    public static void main(String[] args) {
        Counter counter = new Counter();
        MyThread t1 = new MyThread(counter);
        MyThread t2 = new MyThread(counter);
        t1.start();
        t2.start();
    }
}
```

In this example, the increment and getCount methods are synchronized to ensure thread-safe access to the count variable.

Inter-thread Communication:

Java provides methods like wait(), notify(), and notifyAll() for inter-thread communication. These methods are used to coordinate the execution of threads.

Example: Producer-Consumer Problem

```
class Storage {
    private int product = 0;

    public synchronized void produce() throws InterruptedException {
        while (product >= 1) {
            wait();
        }
        product++;
        System.out.println("Produced: " + product);
        notifyAll();
    }

    public synchronized void consume() throws InterruptedException {
        while (product <= 0) {
            wait();
        }
        product--;
        System.out.println("Consumed: " + product);
        notifyAll();
    }
}

class Producer extends Thread {
    Storage storage;
```

```
    public Producer(Storage storage) {
        this.storage = storage;
    }

    public void run() {
        try {
            for (int i = 0; i < 5; i++) {
                storage.produce();
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class Consumer extends Thread {
    Storage storage;

    public Consumer(Storage storage) {
        this.storage = storage;
    }

    public void run() {
        try {
            for (int i = 0; i < 5; i++) {
                storage.consume();
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Storage storage = new Storage();
        Producer producer = new Producer(storage);
        Consumer consumer = new Consumer(storage);
        producer.start();
        consumer.start();
    }
}
```

In this example, the producer and consumer threads coordinate using wait() and notify() to manage the production and consumption of goods in a shared storage.

6.3 Deadlock

Deadlock occurs when two or more threads are blocked forever because they are waiting for each other to release resources. This is typically caused by circular dependencies between threads holding and requesting resources.

Example of Deadlock:

```
class ThreadA extends Thread {
    private final Object lock1;
    private final Object lock2;

    public ThreadA(Object lock1, Object lock2) {
        this.lock1 = lock1;
        this.lock2 = lock2;
    }

    public void run() {
        synchronized (lock1) {
            System.out.println("ThreadA: Holding lock1...");
            try { Thread.sleep(100); } catch (InterruptedException e) {}
            synchronized (lock2) {
                System.out.println("ThreadA: Holding lock2...");
            }
        }
    }
}

class ThreadB extends Thread {
    private final Object lock1;
    private final Object lock2;

    public ThreadB(Object lock1, Object lock2) {
        this.lock1 = lock1;
        this.lock2 = lock2;
    }

    public void run() {
        synchronized (lock2) {
            System.out.println("ThreadB: Holding lock2...");
            try { Thread.sleep(100); } catch (InterruptedException e) {}
            synchronized (lock1) {
                System.out.println("ThreadB: Holding lock1...");
            }
```

```
        }
    }
}

public class DeadlockExample {
    public static void main(String[] args) {
        Object lock1 = new Object();
        Object lock2 = new Object();

        ThreadA t1 = new ThreadA(lock1, lock2);
        ThreadB t2 = new ThreadB(lock1, lock2);
        t1.start();
        t2.start();
    }
}
```

In this case, ThreadA holds lock1 and waits for lock2, while ThreadB holds lock2 and waits for lock1, causing a deadlock.

Preventing Deadlock:

1. **Avoid Nested Locks**: Avoid acquiring multiple locks at the same time if possible.
2. **Lock Ordering**: Always acquire locks in a fixed order.

6.4 Thread Group

A thread group is a mechanism for managing groups of threads. It allows you to perform operations like interrupting multiple threads at once or checking the status of multiple threads.

Example: Using a Thread Group

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

    public static void main(String[] args) {
        ThreadGroup group = new ThreadGroup("MyThreadGroup");

        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.setThreadGroup(group);
        t2.setThreadGroup(group);

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

```
    }
}
```

In this example, two threads t1 and t2 are assigned to the same thread group MyThreadGroup.

6.5 Unit Summary

In this unit, we explored the concept of multithreaded programming in Java, focusing on key elements like thread priorities, synchronization, inter-thread communication, and deadlocks. We learned how to manage thread execution, ensure safe interaction between threads, and prevent common issues like deadlock. Additionally, we examined the thread group concept for managing multiple threads as a single unit.

6.6 Check Your Progress

1. **What is the purpose of thread synchronization in Java?**
    - Answer: Synchronization ensures that only one thread at a time can access a resource, preventing race conditions and ensuring thread safety.
2. **What is a deadlock? Provide an example of how it can occur in a multithreaded environment.**
    - Answer: A deadlock is a situation where two or more threads are blocked forever because they are waiting for each other to release resources. An example occurs when two threads hold locks on different resources and are each waiting for the other to release their lock.
3. **How can thread priorities influence the execution of threads in Java?**
    - Answer: Thread priorities affect the order in which threads are executed. A thread with a higher priority may be executed before a thread with a lower priority, although the actual behavior depends on the operating system's thread scheduler.

Multiple Choice Questions (MCQs)

1. **What is the primary purpose of multithreading in Java?**
    - A) To increase the size of the program
    - B) To make the program slower
    - C) To improve the performance by allowing concurrent execution
    - D) To make the program less responsive
    - **Answer: C) To improve the performance by allowing concurrent execution**
2. **Which method is used to start a thread in Java?**
    - A) startThread()
    - B) begin()
    - C) run()
    - D) start()
    - **Answer: D) start()**
3. **Which of the following is true about thread priorities in Java?**
    - A) Thread priority affects the order in which threads are executed
    - B) Thread priority is ignored by the Java Virtual Machine (JVM)

- C) All threads have the same priority by default
- D) Thread priority can be set only by system threads
- **Answer: A) Thread priority affects the order in which threads are executed**

4. **Which keyword is used for thread synchronization in Java?**
   - A) synchronized
   - B) concurrent
   - C) lock
   - D) multithread
   - **Answer: A) synchronized**
5. **What can lead to a deadlock in multithreaded programming?**
   - A) Threads running in parallel
   - B) Mutual dependency of two or more threads on resources
   - C) High priority threads
   - D) Using the synchronized keyword
   - **Answer: B) Mutual dependency of two or more threads on resources**
6. **What is the function of the Thread.sleep() method?**
   - A) Pauses the execution of a thread indefinitely
   - B) Pauses the thread for a specified time
   - C) Stops the thread permanently
   - D) Starts a new thread
   - **Answer: B) Pauses the thread for a specified time**
7. **Which class is used to create a thread in Java by implementing the Runnable interface?**
   - A) Thread
   - B) Runnable
   - C) Task
   - D) Executor
   - **Answer: A) Thread**
8. **Which of the following is the default priority of a thread in Java?**
   - A) MIN_PRIORITY
   - B) MAX_PRIORITY
   - C) NORM_PRIORITY
   - D) 0
   - **Answer: C) NORM_PRIORITY**
9. **Which method is used to stop a thread in Java?**
   - A) stop()
   - B) terminate()
   - C) halt()
   - D) There is no direct method to stop a thread
   - **Answer: D) There is no direct method to stop a thread**
10. **Which Java class is used to group multiple threads?**
    - A) ThreadGroup
    - B) ThreadPool
    - C) ExecutorService
    - D) RunnableGroup
    - **Answer: A) ThreadGroup**

True/False Questions

1. **Thread priorities in Java can be set using the setPriority() method.**
   - **True**
2. **In Java, deadlock occurs when two or more threads are waiting on each other to release resources that they hold.**
   - **True**
3. **Synchronization in Java is a technique that prevents multiple threads from accessing the same resource at the same time.**
   - **True**
4. **A thread cannot be started more than once in Java after it has completed execution.**
   - **True**
5. **In Java, a thread group allows you to group threads to manage them collectively.**
   - **True**

Fill in the Blanks

1. **In Java, a thread can be put into a blocked state when it is waiting for a _____ to become available.**
   - **Answer: resource**
2. **The _____ method is used to stop a thread temporarily, without terminating it.**
   - **Answer: sleep()**
3. **In Java, _____ is the concept where two or more threads are blocked forever because they are each waiting for the other to release a resource.**
   - **Answer: deadlock**
4. **To synchronize a method in Java, the _____ keyword is used.**
   - **Answer: synchronized**
5. **The default priority of a thread in Java is _____.**
   - **Answer: NORM_PRIORITY**

Unit 7: **I/O Basics**

7.1 Introduction

7.2 Streams, the stream classes, the predefined streams

7.3 Reading console input, writing console output

7.4 The transient and volatile modifiers, using instance of native methods

7.5 Unit Summary

7.6 Check your progress

7.1 Introduction

In this unit, we will explore the fundamentals of Input and Output (I/O) in programming, particularly focusing on the Java language. I/O operations are essential for reading from and writing to various sources, such as the console, files, and network connections. Understanding streams and how to handle them will enable you to manage data efficiently in your applications.

Key Concepts:

- Input and Output operations
- Streams and their role in I/O
- How data is transferred in different directions (input vs. output)

7.2 Streams, the Stream Classes, the Predefined Streams

What is a Stream? In Java, a stream represents a sequence of data that can be read or written to. Streams abstract the data flow, allowing developers to work with I/O operations without dealing directly with the underlying hardware.

Stream Classes in Java:

- **InputStream**: For reading byte data.
  - Example: FileInputStream, BufferedInputStream
- **OutputStream**: For writing byte data.
  - Example: FileOutputStream, BufferedOutputStream
- **Reader**: For reading character data.
  - Example: FileReader, BufferedReader
- **Writer**: For writing character data.
  - Example: FileWriter, BufferedWriter

Predefined Streams: Java provides several predefined streams for convenient I/O operations:

- **System.in**: Standard input stream (usually the keyboard).
- **System.out**: Standard output stream (usually the console).
- **System.err**: Standard error output stream (used for error messages).

7.3 Reading Console Input, Writing Console Output

Reading Console Input: To read data from the user via the console, Java provides the Scanner class. This class allows you to read various types of data, such as strings, integers, and more.

- Example:

```
Scanner scanner = new Scanner(System.in);
System.out.print("Enter your name: ");
String name = scanner.nextLine();
```

Writing Console Output: Java allows you to write to the console using System.out.println() or System.out.print().

- Example: `System.out.println("Hello, " + name + "!");`

In more complex scenarios, you may use PrintWriter for better formatting and handling of output, especially for files or network-based I/O.

7.4 The Transient and Volatile Modifiers, Using Instance of Native Methods

The transient Modifier: The transient keyword in Java is used to indicate that a field should not be serialized. Serialization is the process of converting an object into a byte stream for storage or transmission. A transient field is ignored during serialization.

- Example: `private transient String password;`

The volatile Modifier: The volatile keyword is used to declare that a variable's value will be modified by multiple threads. This ensures that the value of the variable is always fetched from the main memory, rather than being cached by individual threads.

- Example: `private volatile boolean flag;`

Native Methods: Java allows you to integrate with native code (like C or C++) using the native keyword. This is useful when you need to access low-level system resources or performance-critical operations that Java cannot handle natively.

- Example: `public native void myNativeMethod();`

7.5 Unit Summary

In this unit, we covered the essential concepts of Java I/O, including:

- Understanding streams and their types
- Using predefined streams like System.in and System.out
- Handling console input and output with classes like Scanner and PrintWriter
- Exploring the use of transient and volatile modifiers
- Understanding native methods and their role in bridging Java with low-level system operations

Mastering these I/O basics is crucial for handling data in real-world applications.

7.6 Check Your Progress

Test your understanding of the concepts covered in this unit. Answer the following questions:

1. What is the difference between InputStream and Reader in Java?
2. Explain the role of System.out in console I/O operations.
3. What is the purpose of the transient keyword in Java?
4. How does the volatile keyword affect multi-threaded programming?
5. What is a native method, and when might you use one in Java?

Practical Exercises:

1. Write a program that reads the user's name and age from the console and prints a message saying "Hello [Name], you are [Age] years old."
2. Create a class with a transient field and demonstrate how serialization works with and without the `transient` modifier.

Multiple-Choice Questions (MCQs)

1. **Which of the following classes is used for reading input from the console in Java?**

   a) FileReader
   b) BufferedReader
   c) PrintWriter
   d) DataInputStream

2. **What is the function of the System.out stream in Java?**

   a) To read input from the user
   b) To write output to the console

c) To manage file input/output
d) To define transient variables

3. **Which modifier ensures that a variable's value is not cached and is read directly from memory every time?**

   a) volatile
   b) transient
   c) static
   d) final

4. **What is the main purpose of the transient keyword in Java?**

   a) To indicate a variable that should not be serialized
   b) To allow a variable to change across different threads
   c) To make a variable static
   d) To indicate a method is native

5. **Which of the following is a predefined stream used to write output to the console in Java?**

   a) System.in
   b) System.out
   c) System.err
   d) Both b and c

True/False Questions

6. **The System.in stream is used to write output to the console.**
   True / False
7. **The volatile modifier can be applied to methods as well as variables in Java.**
   True / False
8. **A variable marked as transient will not be serialized during object serialization.**
   True / False
9. **BufferedReader is typically used for reading input from a file, not from the console.**
   True / False
10. **The native keyword in Java is used to define methods that are implemented in other languages like C or C++.**
    True / False

Fill in the Blanks

11. The _____ modifier ensures that a variable's value is always read from memory, ensuring it is not cached in multi-threaded environments.
12. The _____ class in Java provides methods to read data from the console.

13. A method that is declared with the _____ modifier indicates that it is implemented in a platform-dependent manner in native code.
14. The predefined stream System.out is used for _____ output in Java.
15. If a field in a class is marked as transient, it will not be _____ during object serialization.

Module III: String handling, Utility classes, java.lang and java.io (12 Hours)

Unit 8: **String handling**

8.1 Introduction
8.2 String constructors
8.3 methods for character extraction
8.4 string searching and comparison
8.5 Data conversion using valueof (), String Buffer
8.6 Unit Summary
8.7 Check your progress

8.1 Introduction to String Handling in Java

Strings are one of the most frequently used objects in Java programming. In Java, a string is an object that represents a sequence of characters. The String class is part of java.lang package and is used to create and manipulate strings. Unlike other objects, strings in Java are immutable, meaning once created, their content cannot be altered.

Key points:

- **Immutability**: Once a string is created, it cannot be modified. Any operation that modifies a string actually creates a new string.
- **String Pool**: Java maintains a special memory area called the String Pool to store string literals. When you create a string using a string literal, Java checks if it already exists in the pool. If it does, the same reference is reused.

8.2 String Constructors

In Java, strings can be created using various constructors. The most commonly used constructors are:

1. **Default constructor**:

   String str = new String();

   This creates an empty string (i.e., "").

2. **Constructor that initializes from a character array**:

   char[] ch = {'H', 'e', 'l', 'l', 'o'};
   String str = new String(ch);

This initializes the string from the given character array.

3. **Constructor from a byte array**:

   ```
   byte[] bytes = {65, 66, 67};
   String str = new String(bytes);
   ```

   This initializes the string from the byte array.

4. **Constructor from another string**:

   ```
   String str = new String("Hello");
   ```

   This creates a new string with the same content as the given string.

8.3 Methods for Character Extraction

Java provides several methods in the String class to extract characters and manipulate strings:

1. **charAt()**: Returns the character at the specified index.

   ```
   String str = "Hello";
   char c = str.charAt(1);  // Returns 'e'
   ```

2. **getChars()**: Copies characters from a string into an array.

   ```
   String str = "Hello";
   char[] arr = new char[5];
   str.getChars(0, 5, arr, 0);  // Copies characters from index 0 to 5
   ```

3. **toCharArray()**: Converts the string into a new character array.

   ```
   String str = "Hello";
   char[] arr = str.toCharArray();  // ['H', 'e', 'l', 'l', 'o']
   ```

8.4 String Searching and Comparison

Java provides several methods for searching within and comparing strings:

1. **indexOf()**: Finds the index of the first occurrence of a specified character or substring.

   ```
   String str = "Hello World";
   int index = str.indexOf('o');  // Returns 4
   ```

2. **lastIndexOf()**: Finds the index of the last occurrence of a specified character or substring.

   ```
   int lastIndex = str.lastIndexOf('o');  // Returns 7
   ```

3. **contains()**: Checks if a substring is present within the string.

   ```
   boolean contains = str.contains("World");  // Returns true
   ```

4. **compareTo()**: Compares two strings lexicographically.

   ```
   String str1 = "apple";
   String str2 = "banana";
   int result = str1.compareTo(str2);  // Negative because "apple" comes before "banana"
   ```

5. **equals()**: Checks if two strings are exactly equal.

   ```
   String str1 = "Hello";
   String str2 = "Hello";
   boolean isEqual = str1.equals(str2);  // Returns true
   ```

6. **equalsIgnoreCase()**: Compares two strings ignoring case.

   ```
   boolean isEqualIgnoreCase = "hello".equalsIgnoreCase("HELLO");  // Returns true
   ```

8.5 Data Conversion Using valueOf(), StringBuffer

Java provides methods for converting various data types to strings and using mutable string representations.

1. **valueOf() Method**: Converts different data types to a string.

   ```
   int x = 100;
   String str = String.valueOf(x);  // Converts integer to string "100"
   ```

2. **StringBuffer**: A mutable sequence of characters. Unlike String, StringBuffer allows modification of string content without creating new objects.

   ```
   StringBuffer buffer = new StringBuffer("Hello");
   buffer.append(" World");  // Modifies the string buffer to "Hello World"
   ```

   Commonly used methods of StringBuffer:

   - append(): Adds a string at the end.
   - insert(): Inserts a string at a specific index.

- delete(): Deletes a substring from the buffer.
- reverse(): Reverses the string.

8.6 Unit Summary

- **Strings in Java** are immutable objects used to represent sequences of characters.
- You can create strings using different constructors, such as the default constructor, from character arrays, byte arrays, or other strings.
- Methods like charAt(), getChars(), and toCharArray() allow you to extract characters from strings.
- String comparison and searching are done using methods like indexOf(), compareTo(), equals(), and contains().
- You can convert data types to strings using valueOf(), and for mutable strings, StringBuffer can be used to perform operations like appending or deleting characters.

8.7 Check Your Progress

1. **What is the difference between String and StringBuffer in Java?**
2. **Explain how to convert an integer to a string using the String.valueOf() method.**
3. **What method would you use to find the index of the first occurrence of a substring within a string?**
4. **What is the output of the following code?**

   ```
   String str = "Hello";
   System.out.println(str.indexOf('e'));
   ```

5. **Write a code to reverse the string "Java Programming" using StringBuffer.**

MCQs:

1. Which of the following is a valid constructor for creating a String object in Java?
   - a) String s = new String("Hello");
   - b) String s = "Hello";
   - c) Both a and b
   - d) None of the above
2. What does the method charAt(int index) of the String class do?
   - a) Returns the ASCII value of the character at the given index
   - b) Returns the character at the given index in the string
   - c) Removes the character at the given index
   - d) None of the above
3. Which method of the String class is used for comparing two strings lexicographically?
   - a) compareTo()
   - b) equals()

    - c) compareToIgnoreCase()
    - d) isEqual()
4. How do you convert an integer to a String in Java?
    - a) String.valueOf(10)
    - b) String.toString(10)
    - c) Integer.toString(10)
    - d) Both a and c
5. Which of the following methods belongs to the StringBuffer class in Java?
    - a) charAt()
    - b) reverse()
    - c) substring()
    - d) toString()
6. What is the result of calling str.indexOf('e') on the string "Hello"?
    - a) 0
    - b) 1
    - c) 4
    - d) -1
7. Which of the following methods can be used to append a string to another in Java?
    - a) String.concat()
    - b) StringBuffer.append()
    - c) StringBuilder.append()
    - d) All of the above
8. The StringBuffer class is _____.
    - a) Immutable
    - b) Mutable
    - c) Thread-safe
    - d) Not thread-safe
9. Which of the following is the correct way to convert a char to a String in Java?
    - a) Character.toString(c)
    - b) String.valueOf(c)
    - c) c.toString()
    - d) Both a and b
10. If you call str.substring(0, 4) on the string "HelloWorld", what will be the output?
    - a) "Hell"
    - b) "Hello"
    - c) "World"
    - d) "HelloWorld"

True/False Questions:

1. The String class is mutable in Java.
    - True / False
2. The StringBuffer class is more efficient than String when making multiple modifications to a string.
    - True / False

3. The charAt() method in Java can be used to extract a character from a string by its index.
    - True / False
4. The compareTo() method is case-sensitive when comparing two strings in Java.
    - True / False
5. The method String.valueOf() can be used to convert an object to a string representation.
    - True / False

Fill in the Blanks:

1. The _____ constructor in Java is used to create a new String object from a character array.
2. The method _____ returns the position of the first occurrence of a specified character in a string.
3. In Java, the _____ method is used to convert any primitive data type to a string.
4. The _____ class is used to perform operations on mutable strings in Java.
5. The method _____ of the StringBuffer class is used to reverse the contents of the buffer.

Unit 9: **Exploring java.lang**

9.1 Introduction

The java.lang package is one of the most essential packages in Java. It provides fundamental classes and interfaces that are used throughout the Java programming language. These classes are automatically imported in every Java program, meaning you don't need to explicitly import the java.lang package.

In this unit, we will explore some of the core components of java.lang that are widely used in Java applications:

- Simple type wrappers
- The System class
- The Class class
- Math functions

By the end of this unit, you'll have a deeper understanding of these classes and how they contribute to Java programming.

9.2 Simple Type Wrappers

In Java, primitive data types (e.g., int, double, char, etc.) are not objects, but sometimes it's necessary to treat them as objects. The java.lang package provides wrapper classes for each primitive data type. These wrapper classes convert primitive types into objects, allowing them to be used in situations where objects are required.

For example:

- Integer (wraps int)
- Double (wraps double)
- Character (wraps char)
- Boolean (wraps boolean)

These classes are useful in scenarios like:

- Working with collections (e.g., ArrayList<Integer>), which only accept objects, not primitives.
- Performing utilities like parsing strings to numeric values (e.g., Integer.parseInt()).

Example:

```
public class WrapperExample {
   public static void main(String[] args) {
      int primitiveInt = 5;
      Integer wrapperInt = Integer.valueOf(primitiveInt); // Boxing

      int unwrappedInt = wrapperInt.intValue(); // Unboxing

      System.out.println("Wrapped Integer: " + wrapperInt);
      System.out.println("Unwrapped Integer: " + unwrappedInt);
   }
}
```

9.3 System Class, Class Class

- **System Class**:
  The System class provides various utility methods related to system-level operations.
  Some of its key features are:
  - Access to system properties and environment variables.
  - Input and output through System.in, System.out, and System.err.
  - Managing the Java runtime environment with methods like System.exit() and System.gc().
  - Accessing the current time through System.currentTimeMillis() and System.nanoTime().

  Example:

  ```
  public class SystemExample {
     public static void main(String[] args) {
        long startTime = System.currentTimeMillis();
        System.out.println("Current time in milliseconds: " + startTime);

        System.out.println("Java version: " + System.getProperty("java.version"));
     }
  }
  ```

- **Class Class**:
  The Class class represents metadata about a class or interface in Java. It allows you to obtain information about the class at runtime, including:
  - Class name
  - Methods, fields, and constructors

- Superclass information
- Interfaces implemented

Example:

```
public class ClassExample {
   public static void main(String[] args) {
      Class<?> cls = String.class;
      System.out.println("Class Name: " + cls.getName());

      System.out.println("Implemented Interfaces:");
      for (Class<?> iface : cls.getInterfaces()) {
         System.out.println(iface.getName());
      }
   }
}
```

9.4 Math Functions

The Math class in the java.lang package provides various mathematical functions and constants. It includes functions for:

- Basic arithmetic operations (Math.addExact(), Math.subtractExact(), etc.)
- Trigonometric functions (Math.sin(), Math.cos(), etc.)
- Logarithmic and exponential functions (Math.log(), Math.exp())
- Random number generation (Math.random())

Example:

```
public class MathExample {
   public static void main(String[] args) {
      double angle = 45.0;
      double radians = Math.toRadians(angle);
      System.out.println("Sin(45°): " + Math.sin(radians));

      double randomValue = Math.random();
      System.out.println("Random Value: " + randomValue);
   }
}
```

9.5 Unit Summary

In this unit, we explored the following important components of the java.lang package:

- **Wrapper classes**: These allow primitive data types to be treated as objects. We reviewed examples of Integer, Double, and other wrapper classes.
- **The System class**: Provides access to system properties, environment variables, and input/output streams. We also explored system-level utilities like System.currentTimeMillis() and System.exit().
- **The Class class**: Enables introspection of class information at runtime. This is useful for reflection-based operations.
- **The Math class**: Provides mathematical functions and constants to perform a variety of calculations and operations.

The java.lang package serves as the backbone for Java applications, providing foundational tools that every Java developer uses.

9.6 Check Your Progress

1. **What are wrapper classes in Java?**
2. **Explain the use of System.currentTimeMillis() and give an example.**
3. **What functionality does the Math.random() method provide?**
4. **How can you obtain information about a class at runtime using the Class class?**
5. **Write a Java program that demonstrates both boxing and unboxing with wrapper classes.**
6. **What is the purpose of System.exit()? Provide an example of its usage.**
7. **What method in the System class can be used to generate a random number between 0 and 1?**
8. **Explain how the Class class can be used to get the name of a class.**

Multiple Choice Questions (MCQs):

1. **Which of the following is NOT a simple type wrapper in java.lang?**
    - a) Integer
    - b) Double
    - c) Character
    - d) String
    - **Answer:** d) String
2. **The System class in Java provides methods for:**
    - a) File operations
    - b) Input/output and memory management
    - c) Command-line arguments handling
    - d) None of the above
    - **Answer:** b) Input/output and memory management
3. **Which method of the Math class returns the absolute value of a number?**
    - a) abs()
    - b) fabs()
    - c) round()
    - d) floor()

- **Answer:** a) abs()

4. **Which class is used to get the runtime environment in Java?**
   - a) Runtime
   - b) System
   - c) Math
   - d) Environment
   - **Answer:** a) Runtime
5. **Which method in the Class class is used to get the name of the class?**
   - a) getName()
   - b) getSimpleName()
   - c) getClassName()
   - d) className()
   - **Answer:** a) getName()
6. **What is the wrapper class for the boolean primitive data type in Java?**
   - a) Boolean
   - b) BooleanWrapper
   - c) Bool
   - d) BooleanType
   - **Answer:** a) Boolean
7. **The Math.sqrt() function in Java returns:**
   - a) The square of a number
   - b) The square root of a number
   - c) The cube root of a number
   - d) None of the above
   - **Answer:** b) The square root of a number
8. **Which class in java.lang allows you to load classes dynamically?**
   - a) ClassLoader
   - b) Class
   - c) System
   - d) Object
   - **Answer:** a) ClassLoader
9. **Which method of the System class is used to exit from a Java program?**
   - a) stop()
   - b) exit()
   - c) terminate()
   - d) end()
   - **Answer:** b) exit()
10. **What is the result of Math.pow(2, 3)?**
    - a) 6
    - b) 8
    - c) 5
    - d) 9
    - **Answer:** b) 8

---

5 Fill in the Blanks:

1. The _____ class in Java provides methods for basic mathematical operations like sqrt(), abs(), and pow().
    - **Answer:** Math
2. The wrapper class for the float primitive data type is _____.
    - **Answer:** Float
3. The _____ class in Java is used to get runtime information and manage memory.
    - **Answer:** System
4. The method _____() in the Class class is used to retrieve the class name.
    - **Answer:** getName
5. The _____ class is used to represent and manipulate characters as objects in Java.
    - **Answer:** Character

True/False Questions:

1. **True or False:** The Class class provides a method forName() to load a class dynamically.
    - **Answer:** True
2. **True or False:** The Math class in Java is a subclass of the Object class.
    - **Answer:** False (It is a final class and does not extend any class.)
3. **True or False:** The System class in Java has a method called currentTimeMillis() that returns the current time in milliseconds.
    - **Answer:** True
4. **True or False:** The Integer wrapper class can be used to convert an integer value into a String object.
    - **Answer:** False (Use String.valueOf(int) or Integer.toString() for this purpose.)
5. **True or False:** The Boolean wrapper class can store both true and false as objects.
    - **Answer:** True

Unit 10: **The utility classes**

10.1 Introduction

10.2 Vector, Stack

10.3 Hash Table, String Tokenizer

10.4 Bit set, Date, Calendar, Gregorian Calendar

10.5 Random, Observable

10.6 Unit Summary

10.7 Check your progress

10.1 Introduction

- **Overview of Utility Classes:**
  - Utility classes in Java are part of the java.util package.
  - These classes provide commonly used functionalities like data structure manipulation, mathematical operations, and handling of system-level features.
  - Examples include Vector, Stack, Hashtable, Date, Calendar, etc.
- **Why Utility Classes Matter:**
  - These classes help in reducing development time and offer predefined solutions to common programming tasks.
  - They improve code efficiency, readability, and maintainability by leveraging well-tested implementations.

10.2 Vector, Stack

- **Vector:**
  - A dynamic array that grows as needed to accommodate new elements.
  - It implements the List interface and is part of java.util.
  - **Key Methods:**
    - add(), remove(), get(), set(), etc.
    - Thread-safe but slower due to synchronization overhead.
  - **Use Cases:**
    - Suitable for storing a resizable array of elements.
- **Stack:**
  - A subclass of Vector, representing a last-in-first-out (LIFO) stack of objects.
  - **Key Methods:**
    - push(), pop(), peek(), empty(), etc.
  - **Use Cases:**
    - Used for managing function calls, undo operations, and navigating web browsers (back and forward).

10.3 Hash Table, String Tokenizer

- **Hashtable:**
  - A collection that stores data in key-value pairs.
  - Implements the Map interface, where each key is unique.
  - **Key Methods:**
    - put(), get(), remove(), containsKey(), etc.
  - **Thread-Safety:**
    - It is synchronized, making it thread-safe but less efficient compared to newer classes like HashMap.

Hash table example:

```
import java.util.Hashtable;

public class HashTableExample {

   public static void main(String[] args) {

      // Create a hashtable object

      Hashtable<String, String> hashtable = new Hashtable<>();



      // Insert key-value pairs into the hashtable

      hashtable.put("1", "One");

      hashtable.put("2", "Two");

      hashtable.put("3", "Three");



      // Retrieve a value using its key

      System.out.println("Key 2: " + hashtable.get("2"));



      // Check if a key exists

      if (hashtable.containsKey("3")) {

         System.out.println("Key 3 is present");
```

```
        }


        // Display all key-value pairs

        hashtable.forEach((key, value) -> System.out.println(key + ": " + value));

    }

}
```

- **StringTokenizer:** StringTokenizer is a legacy class used for splitting strings into tokens.
    - A utility class used to break a string into tokens (substrings) based on specified delimiters.
    - **Key Methods:**
        - hasMoreTokens(), nextToken(), countTokens().
    - **Use Cases:**
        - Parsing strings where the delimiters are known.

StringTokenizer example:

```
import java.util.StringTokenizer;

public class StringTokenizerExample {

    public static void main(String[] args) {

        String text = "Java is fun, easy to learn!";

        StringTokenizer tokenizer = new StringTokenizer(text, " ,!");

// Delimiters: space, comma, exclamation

        // Tokenize and print each token

        while (tokenizer.hasMoreTokens()) {

            System.out.println(tokenizer.nextToken());

        }    } }
```

10.4 Bit Set, Date, Calendar, Gregorian Calendar

- **BitSet:**
  - A class representing a set of bits (binary values), allowing efficient manipulation of bits.
  - **Key Methods:**
    - set(), clear(), get(), and(), or().
  - **Use Cases:**
    - Used for bit-level operations like flags, masks, and binary calculations.

```
import java.util.BitSet;

public class BitSetExample {

   public static void main(String[] args) {

      BitSet bitSet = new BitSet();

      // Set some bits

      bitSet.set(0);

      bitSet.set(3);

      bitSet.set(5);

      // Print the bit set

      System.out.println("BitSet: " + bitSet);

      // Check if a specific bit is set

      System.out.println("Is bit 3 set? " + bitSet.get(3));

   }

}
```

- **Date:**
  - Represents a specific point in time, with both date and time information.
  - **Key Methods:**
    - getTime(), setTime(), before(), after(), etc.
  - **Deprecation:**
    - Some methods are deprecated, with Calendar and LocalDate being recommended alternatives.

```
import java.util.Date;

public class DateExample {

    public static void main(String[] args) {

        // Create a new Date object

        Date date = new Date();

        // Print the current date and time

        System.out.println("Current Date and Time: " + date.toString());

    }

}
```

- **Calendar:**
  - A more flexible class for handling dates and times.
  - Provides methods for date manipulation and calculations.
  - **Key Methods:**
    - get(), set(), add(), getInstance(), etc.
  - **Use Cases:**
    - Complex date and time arithmetic, such as adding months or days.

```
import java.util.Calendar;

public class CalendarExample {

    public static void main(String[] args) {

        Calendar calendar = Calendar.getInstance();

        // Get current date and time

        System.out.println("Current Date and Time: " + calendar.getTime());

        // Set a specific date

        calendar.set(2024, Calendar.NOVEMBER, 30);

        System.out.println("Set Date: " + calendar.getTime());

    }     }
```

- **GregorianCalendar:**
  - A subclass of Calendar that implements the Gregorian calendar system.
  - **Use Cases:**
    - Handling dates that comply with the Gregorian calendar (standard calendar used worldwide).

```
import java.util.GregorianCalendar;

public class GregorianCalendarExample {

    public static void main(String[] args) {

        GregorianCalendar calendar = new GregorianCalendar();

        // Get current date and time in Gregorian format

        System.out.println("Current Date: " + calendar.getTime());

        // Set a specific date

        calendar.set(2024, Calendar.DECEMBER, 25);

        System.out.println("Set Date: " + calendar.getTime());

    }

}
```

10.5 Random, Observable

- **Random:**
  - A utility class for generating random numbers.
  - **Key Methods:**
    - nextInt(), nextDouble(), nextBoolean(), nextFloat().
  - **Use Cases:**
    - Used in games, simulations, and random data generation.

```
import java.util.Random;

public class RandomExample {

    public static void main(String[] args) {

        Random random = new Random();
```

```
        // Generate random integer
        int randInt = random.nextInt(100);  // Random number between 0 and 99
        System.out.println("Random Integer: " + randInt);
        // Generate random boolean
        boolean randBool = random.nextBoolean();
        System.out.println("Random Boolean: " + randBool);
    }
}
```

- **Observable:**
  - Represents an object that can be observed by other objects (observers).
  - The Observable class is used with the **Observer design pattern**.
  - **Key Methods:**
    - addObserver(), deleteObserver(), deleteObservers(), setChanged(), etc.
  - **Use Cases:**
    - Useful in scenarios where one object needs to notify other objects about state changes.

```
import java.util.Observable;
import java.util.Observer;
class WeatherStation extends Observable {
    private String weatherCondition;

    public void setWeatherCondition(String condition) {
        this.weatherCondition = condition;
        setChanged();  // Mark as changed
        notifyObservers(condition);  // Notify observers
    }
}
```

```
class WeatherDisplay implements Observer {
    private String currentCondition;
    @Override
    public void update(Observable o, Object arg) {
        this.currentCondition = (String) arg;
        System.out.println("Weather updated: " + currentCondition);
    }
}
public class ObservableExample {
    public static void main(String[] args) {
        WeatherStation station = new WeatherStation();
        WeatherDisplay display = new WeatherDisplay();
        // Add observer to the observable
        station.addObserver(display);
        // Change weather condition and notify observers
        station.setWeatherCondition("Sunny");
        station.setWeatherCondition("Rainy");
    }
}
```

10.6 Unit Summary

- **Key Takeaways:**
  - Utility classes in java.util package provide efficient, reusable solutions for common tasks.

- Classes like Vector, Stack, Hashtable, BitSet, Random, and Observable offer data structures, utilities for random numbers, and methods for time and date manipulation.
- Understanding when and how to use these classes can lead to more efficient, maintainable, and organized code.
- Modern alternatives to older classes (like HashMap, ArrayList, and LocalDate) may offer better performance and flexibility.

10.7 Check Your Progress

- **Review Questions:**
  1. What is the difference between Vector and Stack?
  2. Explain the thread-safety feature of Hashtable.
  3. How does StringTokenizer work, and when should it be used?
  4. What are the advantages of using BitSet over an array of booleans?
  5. Describe the role of GregorianCalendar in date manipulation.
  6. How does the Random class generate random numbers?
  7. What is the purpose of the Observable class in Java, and how does it work?
- **Practical Tasks:**
  1. Create a simple program that uses Vector to store a list of integers and prints them.
  2. Implement a Stack that mimics the behavior of undo/redo functionality.
  3. Write a program that uses HashTable to store employee names with their ID numbers.
  4. Use StringTokenizer to parse a comma-separated list of words.
  5. Experiment with Random to generate a series of random integers and floats.

Multiple Choice Questions (MCQs)

1. **Which of the following interfaces does a Vector class implement in Java?**
   - a) Set
   - b) List
   - c) Map
   - d) Queue
2. **What is the initial capacity of a Stack in Java, if not specified?**
   - a) 10
   - b) 20
   - c) 50
   - d) 30
3. **Which class does the HashTable class extend in Java?**
   - a) Object
   - b) AbstractMap
   - c) HashMap
   - d) Dictionary
4. **Which method is used to split a string into tokens using the StringTokenizer class?**

- a) next()
- b) nextToken()
- c) split()
- d) getNext()

5. **Which class is used to store a collection of bits in Java?**
   - a) BitSet
   - b) BitArray
   - c) Set
   - d) HashSet
6. **Which method is used to get the current time in Date class?**
   - a) getCurrentTime()
   - b) getDate()
   - c) now()
   - d) getTime()
7. **Which method in Calendar is used to get the current year?**
   - a) getYear()
   - b) get(Calendar.YEAR)
   - c) currentYear()
   - d) year()
8. **Which class is an extension of Calendar and provides a more modern approach to handling dates?**
   - a) GregorianCalendar
   - b) TimeCalendar
   - c) DateTime
   - d) TimeZone
9. **The Random class in Java is used to generate:**
   - a) Secure random numbers
   - b) Random booleans
   - c) Random numbers
   - d) Hash codes
10. **Which class allows an object to observe changes to another object?**
    - a) Random
    - b) Calendar
    - c) Observable
    - d) Date

True/False Questions

1. **A Stack in Java is a part of the Collection Framework.**
2. **In Java, the StringTokenizer class can break a string into tokens using a specified delimiter.**
3. **A Vector in Java is thread-safe, meaning it can be safely used by multiple threads at the same time.**
4. **HashTable is an unordered collection of key-value pairs in Java.**

5. **The Random class in Java cannot generate random numbers for all primitive data types.**

Fill in the Blanks

1. The _____ class in Java allows us to represent a collection of bits, and we can manipulate them individually.
2. A _____ is a last-in, first-out (LIFO) data structure in Java that extends Vector.
3. The _____ method of the Calendar class is used to retrieve a specific field such as year, month, or day.
4. The _____ class is an extension of Calendar that handles date and time calculations based on the Gregorian calendar.
5. The _____ class in Java is used to generate random numbers and can be seeded for repeatable results.

Unit 11: **Input / Output**

11.1 Introduction

11.2 Exploring java.io: The java.io classes and interface

11.3 File class and methods for creating, renaming, listing and deleting files and directories

11.4 I/O stream classes (File Input Sream, File Output Stream, Buffered Input Stream, Buffered Output Stream , Push Back Input Stream, Input Stream Reader, Buffered Reader, Buffered Writer, Print Stream, Random Access File)

11.5 Unit Summary

11.6 Check your progress

11.1 Introduction

- **Overview of Input/Output (I/O):**
  - **I/O** in programming refers to the process of reading data from or writing data to an external source such as a file, network, or database.
  - In Java, I/O operations are handled using **streams** and **files**.
  - **Stream** is a sequence of data elements that are read from or written to an I/O device (like a file or console).
- **Types of I/O:**
  - **Byte Stream:** Handles raw binary data (used for image, audio, etc.).
  - **Character Stream:** Handles text data and uses Unicode (used for reading/writing text files).
- **Why Java I/O?**
  - Java provides a rich set of classes to handle I/O, allowing reading from and writing to different data sources with ease.

11.2 Exploring java.io: The java.io classes and interface

- **java.io package:**
  - The java.io package provides classes for system input and output through data streams, file system handling, and the use of readers and writers for text-based data.
  - Some important classes and interfaces in the java.io package include:
    - **File** – Represents file and directory pathnames.
    - **InputStream / OutputStream** – Base classes for reading and writing byte data.
    - **Reader / Writer** – Base classes for handling character data.
    - **FileReader / FileWriter** – Specialized classes for file reading/writing using characters.
- **Example:**

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

```
public class FileExample {
   public static void main(String[] args) {
      File file = new File("example.txt");
      if (file.exists()) {
         System.out.println("File exists!");
      } else {
         System.out.println("File does not exist.");
      }
   }
}
```

11.3 File class and methods for creating, renaming, listing and deleting files and directories

- **The File class:**
  - The File class represents file and directory pathnames in an abstract manner, providing methods for file manipulation.
- **Methods of the File class:**
  - **createNewFile()** – Creates a new file.
  - **delete()** – Deletes a file or directory.
  - **renameTo(File dest)** – Renames a file or directory.
  - **list()** – Returns an array of file names in a directory.
  - **mkdir() / mkdirs()** – Creates directories (single or nested).
- **Example:**

```
import java.io.*;

public class FileOperations {
   public static void main(String[] args) throws IOException {
      File file = new File("sample.txt");
      if (!file.exists()) {
         file.createNewFile();
         System.out.println("File created.");
      }

      // Renaming the file
      File newFile = new File("new_sample.txt");
      file.renameTo(newFile);
      System.out.println("File renamed.");

      // Listing files in a directory
      File dir = new File(".");
      String[] files = dir.list();
      for (String filename : files) {
         System.out.println(filename);
      }
```

```
        // Deleting the file
        newFile.delete();
        System.out.println("File deleted.");
    }
}
```

11.4 I/O stream classes

Java provides a wide variety of I/O stream classes for reading and writing data. Here are some of the commonly used classes:

1. **FileInputStream / FileOutputStream**
    - Used for reading and writing bytes to/from files.
    - **Example:**

```
import java.io.*;

public class FileStreamExample {
    public static void main(String[] args) throws IOException {
        FileInputStream input = new FileInputStream("input.txt");
        int data;
        while ((data = input.read()) != -1) {
            System.out.print((char) data);
        }
        input.close();
    }
}
```

2. **BufferedInputStream / BufferedOutputStream**
    - Provides efficient reading and writing by buffering data.
    - **Example:**

```
import java.io.*;

public class BufferedStreamExample {
    public static void main(String[] args) throws IOException {
        BufferedInputStream input = new BufferedInputStream(new
FileInputStream("input.txt"));
        int data;
        while ((data = input.read()) != -1) {
            System.out.print((char) data);
        }
        input.close();
    }
}
```

3. **PushBackInputStream**
    - Allows the "pushing back" of bytes for re-reading.
4. **InputStreamReader / BufferedReader**
    - **InputStreamReader** bridges byte streams and character streams.
    - **BufferedReader** reads text efficiently by buffering the input.
    - **Example:**

```
import java.io.*;

public class BufferedReaderExample {
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
        System.out.println("Enter some text: ");
        String input = reader.readLine();
        System.out.println("You entered: " + input);
    }
}
```

5. **BufferedWriter / PrintWriter**
    - BufferedWriter writes text efficiently, while PrintWriter is used for formatted printing.
    - **Example:**

```
import java.io.*;

public class PrintWriterExample {
    public static void main(String[] args) throws IOException {
        PrintWriter writer = new PrintWriter(new BufferedWriter(new
FileWriter("output.txt")));
        writer.println("Hello, world!");
        writer.close();
    }
}
```

6. **RandomAccessFile**
    - Allows random access to a file, reading and writing at any position.
    - **Example:**

```
import java.io.*;

public class RandomAccessFileExample {
    public static void main(String[] args) throws IOException {
        RandomAccessFile file = new RandomAccessFile("data.txt", "rw");
        file.writeInt(12345);
        file.seek(0);  // Move to the beginning of the file
```

```
            System.out.println(file.readInt());  // Reads 12345
            file.close();
        }
    }
```

11.5 Unit Summary

- **I/O Streams** are fundamental for file handling, data transmission, and interaction with the external environment.
- Java provides multiple classes for handling files and streams, categorized into byte-based and character-based classes.
- Efficient reading and writing can be achieved through buffering and specific classes like BufferedReader, BufferedWriter, and PrintWriter.
- The **File** class is essential for file creation, deletion, renaming, and listing contents.

11.6 Check your progress

1. **Explain the difference between FileInputStream and BufferedInputStream.**
   - FileInputStream reads bytes directly from a file, while BufferedInputStream provides a buffer to improve performance during file reading.
2. **What does the RandomAccessFile class allow you to do?**
   - It allows reading and writing data at any point within a file, enabling random access to the file contents.
3. **How do you create a new directory in Java using the File class?**
   - Use the mkdir() or mkdirs() methods of the File class to create directories.

Multiple Choice Questions (MCQs)

1. **Which of the following class is used to create, rename, and delete files in Java?**
   a) java.util.File
   b) java.io.File
   c) java.nio.File
   d) java.io.Path
   Answer: b
2. **Which method in the File class is used to create a new file?**
   a) createNewFile()
   b) createFile()
   c) makeFile()
   d) newFile()
   Answer: a
3. **What is the main advantage of Buffered Streams over File Streams?**
   a) They use less memory
   b) They provide faster data access by reducing I/O operations
   c) They simplify file handling code

d) They do not require closing
Answer: b

4. **Which stream class allows random access to a file in Java?**
   a) FileInputStream
   b) FileOutputStream
   c) RandomAccessFile
   d) PrintStream
   Answer: c
5. **What is the purpose of the PushbackInputStream class?**
   a) To push data into a buffer for writing
   b) To allow unread bytes to be pushed back into the input stream
   c) To reverse the order of input data
   d) To improve reading speed
   Answer: b
6. **Which method of BufferedReader is used to read a line of text?**
   a) read()
   b) readLine()
   c) readText()
   d) readAll()
   Answer: b
7. **What is the default size of the buffer used in a BufferedInputStream?**
   a) 1024 bytes
   b) 8192 bytes
   c) 4096 bytes
   d) 2048 bytes
   Answer: b
8. **Which of the following classes is used to write formatted data to an output stream?**
   a) BufferedWriter
   b) PrintStream
   c) InputStreamReader
   d) RandomAccessFile
   Answer: b
9. **How do you delete a directory using the File class?**
   a) deleteDirectory()
   b) remove()
   c) delete()
   d) deleteDir()
   Answer: c
10. **Which of the following is not an InputStream class?**
    a) FileInputStream
    b) BufferedInputStream
    c) PushbackInputStream
    d) PrintStream
    Answer: d

True/False Questions

1. The File class can be used to check if a file exists.
   Answer: True
2. The BufferedWriter class is used to read files efficiently.
   Answer: False (It is used to write files.)
3. The RandomAccessFile class allows both read and write access to a file.
   Answer: True
4. The delete() method of the File class can delete non-empty directories.
   Answer: False
5. The FileInputStream class is used for reading binary data from a file.
   Answer: True

Fill in the Blanks

1. The _____ class in java.io is used to handle file and directory operations.
   Answer: File
2. To read text data efficiently, the _____ class is used.
   Answer: BufferedReader
3. The _____ stream allows random access to a file for both reading and writing.
   Answer: RandomAccessFile
4. The _____ class allows you to write formatted text to an output stream.
   Answer: PrintStream
5. The method _____ of the File class can be used to rename a file or directory.
   Answer: renameTo

Module IV: Networking, Images, Applet class and Swing (12 Hours)
Unit 12: **Networking**
12.1 Introduction
12.2 Socket overview: Stream Sockets, Datagram sockets
12.3 Manipulating URLs, Establishing a simple Server/Client using Stream Sockets
12.4 Connectionless Client/Server Interaction with Datagrams
12.5 Unit Summary
12.6 Check your progress

12.1 Introduction

Networking in Java is a powerful feature that enables communication between systems. Java provides a robust API for network programming, allowing developers to create applications that can send and receive data across networks. This unit introduces key concepts and practical implementations of networking in Java, focusing on sockets, URLs, and client/server models.

12.2 Socket Overview: Stream Sockets, Datagram Sockets

Sockets:
A socket is an endpoint for communication between two machines. In Java, sockets facilitate two-way communication over networks.

1. **Stream Sockets (TCP):**
   - Reliable, connection-oriented communication.
   - Ensures data is received in the correct order.
   - Example: Web browsing, email protocols.
2. **Datagram Sockets (UDP):**
   - Connectionless, lightweight communication.
   - No guarantee of delivery or order.
   - Example: Online gaming, video streaming.

Java provides classes such as Socket and ServerSocket for TCP and DatagramSocket for UDP.

12.3 Manipulating URLs, Establishing a Simple Server/Client using Stream Sockets

1. **Manipulating URLs:**
   Java's java.net.URL class allows working with Uniform Resource Locators (URLs). Key methods include:

- getProtocol(): Returns the protocol (e.g., HTTP, FTP).
- getHost(): Retrieves the hostname.
- getPort(): Returns the port number.
- getPath(): Retrieves the file path.

Example:

```
import java.net.URL;

public class URLExample {
   public static void main(String[] args) throws Exception {
      URL url = new URL("https://example.com:8080/index.html");
      System.out.println("Protocol: " + url.getProtocol());
      System.out.println("Host: " + url.getHost());
      System.out.println("Port: " + url.getPort());
      System.out.println("Path: " + url.getPath());
   }
}
```

2. **Establishing a Simple Server/Client using Stream Sockets:**
   A typical TCP server and client communicate through sockets.

   Server Code Example:

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

public class SimpleServer {
   public static void main(String[] args) throws IOException {
      ServerSocket serverSocket = new ServerSocket(5000);
      System.out.println("Server is listening...");
      Socket socket = serverSocket.accept();
      BufferedReader input = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
      PrintWriter output = new PrintWriter(socket.getOutputStream(), true);
      String message = input.readLine();
      System.out.println("Client says: " + message);
      output.println("Hello, Client!");
      socket.close();
      serverSocket.close();
   }
}
```

   Client Code Example:

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

```
import java.net.*;

public class SimpleClient {
   public static void main(String[] args) throws IOException {
      Socket socket = new Socket("localhost", 5000);
      PrintWriter output = new PrintWriter(socket.getOutputStream(), true);
      BufferedReader input = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
      output.println("Hello, Server!");
      System.out.println("Server says: " + input.readLine());
      socket.close();
   }
}
```

12.4 Connectionless Client/Server Interaction with Datagrams

Datagram communication is connectionless and uses the DatagramSocket and DatagramPacket classes in Java.

Server Code Example:

```
import java.net.*;
public class DatagramServer {
   public static void main(String[] args) throws Exception {
      DatagramSocket serverSocket = new DatagramSocket(5000);
      byte[] buffer = new byte[1024];
      DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
      System.out.println("Server is waiting for a message...");
      serverSocket.receive(packet);
      String message = new String(packet.getData(), 0, packet.getLength());
      System.out.println("Received: " + message);
      serverSocket.close();
   }
}
```

Client Code Example:

```
import java.net.*;
public class DatagramClient {
   public static void main(String[] args) throws Exception {
      DatagramSocket clientSocket = new DatagramSocket();
      String message = "Hello, Datagram Server!";
      byte[] buffer = message.getBytes();
      InetAddress serverAddress = InetAddress.getByName("localhost");
      DatagramPacket packet = new DatagramPacket(buffer, buffer.length, serverAddress, 5000);
      clientSocket.send(packet);
```

```
        System.out.println("Message sent to server.");
        clientSocket.close();
    }
}
```

12.5 Unit Summary

In this unit, we explored Java's networking capabilities:

- **Stream Sockets (TCP):** Reliable and connection-oriented communication for client/server models.
- **Datagram Sockets (UDP):** Lightweight, connectionless communication for fast data exchange.
- **URL Manipulation:** Extracting and using components of a URL with the URL class.
- Practical examples to implement server/client communication using both TCP and UDP.

12.6 Check Your Progress

1. What is the difference between TCP and UDP?
2. Write a program to extract components of a URL in Java.
3. Implement a simple server and client using ServerSocket and Socket classes.
4. Describe a scenario where UDP is preferable over TCP.
5. Explain the purpose of the DatagramPacket class.

Multiple Choice Questions (MCQs)

1. **What is the main purpose of the Socket class in Java?**
    - A. To establish a connection between two computers
    - B. To store data locally
    - C. To read and write files
    - D. To manage threads

    Answer: A
2. **Which of the following is true about stream sockets?**
    - A. They use TCP for communication.
    - B. They are used for connectionless communication.
    - C. They work only within a single machine.
    - D. They cannot handle multiple connections.

    Answer: A
3. **Which method is used to open a connection to a URL in Java?**
    - A. connect()
    - B. openConnection()
    - C. getInputStream()
    - D. readLine()

    Answer: B

4. **Datagram sockets are associated with which protocol?**
    - A. TCP
    - B. HTTP
    - C. UDP
    - D. FTP

    Answer: C
5. **In a connectionless communication model, data is transmitted using:**
    - A. Persistent connections
    - B. Data streams
    - C. Packets
    - D. File transfer protocols

    Answer: C
6. **Which of the following is NOT a method of the Socket class?**
    - A. getOutputStream()
    - B. close()
    - C. bind()
    - D. connect()

    Answer: C
7. **What is the key difference between stream sockets and datagram sockets?**
    - A. Stream sockets use UDP, datagram sockets use TCP.
    - B. Stream sockets use TCP, datagram sockets use UDP.
    - C. Both use TCP but have different connection methods.
    - D. Stream sockets are slower than datagram sockets.

    Answer: B
8. **Which class is used to manipulate URLs in Java?**
    - A. URLConnection
    - B. URL
    - C. Socket
    - D. InetAddress

    Answer: B
9. **Which method sends data in a DatagramSocket?**
    - A. sendPacket()
    - B. sendData()
    - C. send()
    - D. send(DatagramPacket p)

    Answer: D
10. **Which Java package contains networking classes?**
    - A. java.util
    - B. java.network
    - C. java.net
    - D. java.io

    Answer: C

True/False Questions

1. Datagram sockets use the TCP protocol for communication.
   Answer: False
2. URLs in Java can be manipulated using the URL class.
   Answer: True
3. A client-server connection using stream sockets requires the server to listen for client connections.
   Answer: True
4. A DatagramSocket provides a connection-oriented communication model.
   Answer: False
5. The InetAddress class is used to represent an IP address in Java networking.
   Answer: True

Fill in the Blanks

1. Stream sockets use the _____ protocol for reliable communication.
   Answer: TCP
2. The class used to represent and manipulate URLs in Java is _____.
   Answer: URL
3. In connectionless communication, data is sent as discrete _____.
   Answer: packets
4. To establish a simple server using stream sockets, the server uses the _____ class in Java.
   Answer: ServerSocket
5. In a DatagramSocket, data is transmitted using _____ packets.
   Answer: UDP

***Unit 13*: Images**

13.1 Introduction

13.2 File formats,

13.3 image fundamentals, creating, loading and displaying images

13.4 ImageObserver, MediaTracker

13.5 Unit Summary

13.6 Check your progress

13.1 Introduction

Images are an integral part of modern applications, enhancing the visual appeal and user experience. Java provides robust tools and libraries for handling images, from creating and loading them to rendering and manipulating them in applications. This unit explores the basics of image handling in Java, including file formats, image fundamentals, and utilities like ImageObserver and MediaTracker.

13.2 File Formats

Java supports various image file formats through its java.awt and javax.imageio packages. Key formats include:

- **JPEG:** Common for photographs; supports lossy compression.
- **PNG:** Supports transparency and lossless compression.
- **GIF:** Useful for animations; supports limited color depth.
- **BMP:** Uncompressed format; rarely used due to large file size.
- **TIFF:** Ideal for professional image processing; supports layers and high color depth.

Java's ImageIO class can read and write most of these formats efficiently, making it versatile for image manipulation tasks.

13.3 Image Fundamentals: Creating, Loading, and Displaying Images

- **Creating Images:** Use Java's BufferedImage class to create new images programmatically.

```
BufferedImage image = new BufferedImage(width, height,
BufferedImage.TYPE_INT_RGB);
Graphics2D g2d = image.createGraphics();
g2d.drawLine(0, 0, width, height); // Example of drawing on the image
```

```
g2d.dispose();
```

- **Loading Images:** Load images from files using ImageIO:

  ```
  BufferedImage img = ImageIO.read(new File("path/to/image.jpg"));
  ```

- **Displaying Images:** Use JPanel or Canvas components to render images in a GUI application:

  ```
  @Override
  protected void paintComponent(Graphics g) {
     super.paintComponent(g);
     g.drawImage(img, 0, 0, null);
  }
  ```

13.4 ImageObserver and MediaTracker

- **ImageObserver Interface:** The ImageObserver interface is used for monitoring the status of image loading. It provides methods to track the progress and react to loading events:

  ```
  public boolean imageUpdate(Image img, int infoflags, int x, int y, int width, int height) {
     return true; // Continue observing
  }
  ```

- **MediaTracker Class:** The MediaTracker class ensures images or media are fully loaded before being used in a program. This avoids rendering incomplete images:

  ```
  MediaTracker tracker = new MediaTracker(this);
  tracker.addImage(image, 0);
  try {
     tracker.waitForAll(); // Wait for all images to load
  } catch (InterruptedException e) {
     e.printStackTrace();
  }
  ```

13.5 Unit Summary

In this unit, you learned:

1. The supported image file formats in Java and their applications.
2. How to create, load, and display images using the BufferedImage and Graphics classes.
3. The roles of ImageObserver and MediaTracker in ensuring efficient image handling and rendering.

13.6 Check Your Progress

1. What are the key differences between JPEG and PNG file formats?
2. Write a code snippet to load and display an image using Java Swing.
3. How does the MediaTracker class handle asynchronous loading of images?
4. Explain the role of the ImageObserver interface in Java applications.

Multiple-Choice Questions (MCQs)

1. **Which Java class is commonly used to load and display images?**
   a) FileReader
   b) ImageIO
   c) MediaTracker
   d) Scanner
   Answer: b) ImageIO
2. **Which image file format supports transparency?**
   a) BMP
   b) JPEG
   c) PNG
   d) TIFF
   Answer: c) PNG
3. **What is the purpose of the ImageObserver interface in Java?**
   a) To observe changes in image size
   b) To track and monitor image loading
   c) To resize images dynamically
   d) To convert images to grayscale
   Answer: b) To track and monitor image loading
4. **Which method is used to draw an image in a Graphics object?**
   a) drawImage()
   b) paintImage()
   c) loadImage()
   d) renderImage()
   Answer: a) drawImage()
5. **What does the MediaTracker class primarily do?**
   a) Converts image formats
   b) Ensures images are loaded before they are displayed
   c) Monitors memory usage of images
   d) Optimizes image rendering
   Answer: b) Ensures images are loaded before they are displayed
6. **Which method in ImageIO is used to read an image from a file?**
   a) readFile()
   b) read()
   c) load()
   d) open()
   Answer: b) read()

7. **JPEG format is typically used for:**
   a) Diagrams with sharp edges
   b) High-compression photographs
   c) Images with transparency
   d) Vector graphics
   Answer: b) High-compression photographs
8. **What happens if an image is not fully loaded when drawImage() is called?**
   a) The program throws an exception
   b) The image is drawn as a blank space
   c) It draws a partial image
   d) Loading halts until the image is fully loaded
   Answer: c) It draws a partial image
9. **Which of these formats is not supported natively by Java's ImageIO?**
   a) BMP
   b) PNG
   c) GIF
   d) WEBP
   Answer: d) WEBP
10. **What is the result of calling ImageIO.write()?**
    a) Saves an image to disk
    b) Reads an image from a stream
    c) Converts an image to grayscale
    d) Resizes the image
    Answer: a) Saves an image to disk

True/False Questions

1. **The MediaTracker class can track the loading of multiple images simultaneously.**
   Answer: True
2. **JPEG format supports both lossless compression and transparency.**
   Answer: False
3. **The drawImage() method requires an ImageObserver as one of its arguments.**
   Answer: True
4. **GIF images cannot be loaded in Java's default libraries.**
   Answer: False
5. **The ImageObserver interface is used to perform operations on an image after it is fully loaded.**
   Answer: True

Fill in the Blanks

1. **The _____ class in Java is used to handle image files like PNG and JPEG.**
   Answer: ImageIO

2. **The method Graphics._____() is used to render images onto a component.**
   Answer: drawImage
3. **_____ is a widely used lossy image format commonly used for photographs.**
   Answer: JPEG
4. **The ImageObserver interface provides notifications when an image's _____ is being loaded.**
   Answer: data
5. **The _____ class ensures that images are fully loaded before being displayed in a Java application.**
   Answer: MediaTracker

Unit 14: **The Applet class**

14.1 Introduction

14.2 applet architecture, passing parameters to applets, getDocumentBase, getCodeBase, and showDocument

14.3 AppletContext and AudioClip interfaces, Graphics class and methods for drawing lines, rectangles, polygons and ovals

14.4 applet architecture, passing parameters to applets, getDocumentBase, getCodeBase, and showDocument,

14.4.1 Swing: Component and Container classes, Layout managers (Flow Layout, Grid Layout, Border Layout), Handling events, Adapter classes, Anonymous inner classes

14.4.2 Swing GUI components: JLabel, JTextField, JTextArea, JButton, JCheckBox, JRadioButton, JList, JComboBox, JScrollBar, JScrollPane, JToolTip, JPanel, JFrame

14.4.3 Menus: JmenuBar, JMenu, JMenuItem, JSeparator

14.5 Unit Summary

14.6 Check your progress

14.1 Introduction

- **Definition of Applets**: Applets are small Java programs that run within a web browser or an applet viewer.
- **Key Characteristics**:
  - Part of the java.applet package.
  - Designed to enhance web pages with interactive capabilities.
  - Executed in a controlled environment (sandbox).
- **Lifecycle of an Applet**:
  - init(): Initializes the applet.
  - start(): Starts or resumes the applet.
  - stop(): Pauses the applet.
  - destroy(): Cleans up resources before termination.

Applet example:

```
import java.applet.Applet;

import java.awt.Graphics;

public class AppletDemo extends Applet {

   public void init() {

      System.out.println("Applet initialized");    }
```

```
    public void start() {
    System.out.println("Applet started");    }
    public void stop() {
    System.out.println("Applet stopped");    }
    public void destroy() {
    System.out.println("Applet destroyed");    }
    public void paint(Graphics g) {
    g.drawString("Hello, Applet!", 50, 50);   }
}
```

Passing Parameters to Applets:

Parameters can be passed using <param> tags in the HTML file.

Example:

HTML code:

```
<applet code="AppletDemo.class" width="300" height="300">
   <param name="message" value="Welcome to Applet!">
</applet>
```

Java Code:

```
public void paint(Graphics g) {
   String message = getParameter("message");
   g.drawString(message, 50, 50);
}
```

14.2 Applet Architecture

- **Structure**: Applet lifecycle methods and their role in execution.
- **Passing Parameters to Applets**:
  - Use <PARAM> tags in the HTML file.
  - Retrieve parameters using getParameter(String name) method.
- **Key Methods**:
  - getDocumentBase(): Returns the URL of the HTML document containing the applet.

- getCodeBase(): Returns the URL of the directory containing the applet's code.
- showDocument(URL url): Requests the browser to display a new page.

getDocumentBase() and getCodeBase()

- getDocumentBase(): Returns the URL of the document containing the applet.
- getCodeBase(): Returns the URL of the applet's code.

Example:

```
import java.applet.Applet;
import java.awt.Graphics;

public class URLDemo extends Applet {
   public void paint(Graphics g) {
      g.drawString("Document Base: " + getDocumentBase(), 10, 20);
      g.drawString("Code Base: " + getCodeBase(), 10, 40);
   }
}
```

---

showDocument

The AppletContext interface is used to interact with the applet's environment, including displaying documents.

Example:

```
import java.applet.*;
import java.net.*;

public class ShowDocumentDemo extends Applet {
   public void start() {
      try {
         URL url = new URL("https://www.example.com");
         getAppletContext().showDocument(url, "_blank");
      } catch (MalformedURLException e) {
         e.printStackTrace();
      }
   }
}
```

14.3 AppletContext and Graphics

- **AppletContext Interface**:
  - Provides methods for applet-to-applet communication.
  - Example methods: getApplet(String name), getAudioClip(URL url).
- **AudioClip Interface**:
  - Methods: play(), loop(), stop() for sound management.

The AppletContext and AudioClip interfaces are part of Java's java.applet package, which is primarily used in Java applet development (now deprecated). Below are practical examples demonstrating their use:

1. AppletContext Interface

The AppletContext interface provides methods for applets to interact with their environment, such as accessing other applets or the document base (URL).

Example:

```
import java.applet.Applet;
import java.applet.AppletContext;
import java.net.URL;

public class AppletContextExample extends Applet {
   @Override
   public void start() {
      AppletContext context = getAppletContext();

      // Show a message in the applet viewer or browser's status bar
      context.showStatus("Applet has started!");

      try {
         // Open a URL in the browser
         URL url = new URL("https://www.example.com");
         context.showDocument(url);
      } catch (Exception e) {
         context.showStatus("Error opening URL: " + e.getMessage());
      }
   }
}
```

Explanation:

- getAppletContext() returns the applet's context.
- showStatus() displays a message in the browser or applet viewer's status bar.
- showDocument(URL) opens the specified URL in the browser.

---

2. AudioClip Interface

The AudioClip interface represents an audio clip that can be played, stopped, or looped.

Example:

```
import java.applet.Applet;
import java.applet.AudioClip;

public class AudioClipExample extends Applet {
    private AudioClip clip;

    @Override
    public void init() {
        // Load an audio clip from the applet's directory
        clip = getAudioClip(getDocumentBase(), "audio.wav");
    }

    @Override
    public void start() {
        // Play the audio clip
        clip.play();
    }

    @Override
    public void stop() {
        // Stop the audio clip
        clip.stop();
    }

    public void loopAudio() {
        // Loop the audio clip
        clip.loop();
    }
}
```

Explanation:

- getAudioClip(URL base, String filename) loads an audio file.
- play() starts playing the audio once.
- loop() plays the audio clip in a loop.
- stop() stops the audio playback.

Notes:

- These APIs were commonly used for applet development but are now considered outdated. Applets were officially deprecated in Java 9 and removed in Java 11.
- For modern applications, consider using JavaFX or external libraries for audio and UI-related tasks.

- **Graphics Class**:
  - Methods for drawing shapes:
    - drawLine(x1, y1, x2, y2)
    - drawRect(x, y, width, height)
    - drawPolygon(int[] xPoints, int[] yPoints, int nPoints)
    - drawOval(x, y, width, height)

Graphics Class and Methods

The Graphics class provides methods to draw shapes.

Example:

```
import java.applet.Applet;
import java.awt.Graphics;

public class GraphicsDemo extends Applet {
   public void paint(Graphics g) {
      g.drawLine(10, 10, 100, 10);         // Draw a line
      g.drawRect(10, 20, 100, 50);        // Draw a rectangle
      g.drawOval(10, 80, 100, 50);        // Draw an oval
      int[] xPoints = {10, 60, 110};
      int[] yPoints = {150, 100, 150};
      g.drawPolygon(xPoints, yPoints, 3);  // Draw a triangle
   }
}
```

14.4 Advanced Topics in Applets

- **14.4.1 Swing**
  - **Component and Container Classes**: Swing components like JComponent and their containers.
  - **Layout Managers**:
    - FlowLayout: Places components sequentially.
    - GridLayout: Organizes components in a grid.

        - BorderLayout: Divides the container into five regions.
    - **Event Handling**:
        - Event listeners like ActionListener, MouseListener.
        - **Adapter Classes**: Simplifies event handling by overriding only required methods.
        - **Anonymous Inner Classes**: Used for inline event handling.
- **14.4.2 Swing GUI Components**
    - **Text Components**:
        - JLabel: Displays static text.
        - JTextField: Single-line editable text field.
        - JTextArea: Multi-line editable text area.
    - **Interactive Components**:
        - JButton: Button for user actions.
        - JCheckBox and JRadioButton: Toggle and radio options.
        - JList and JComboBox: List and dropdown menu.
    - **Scroll and Panel Components**:
        - JScrollBar: Adds scrollbars to components.
        - JScrollPane: Provides scrolling view of content.
        - JPanel: Groups components.
    - **Main Frame**:
        - JFrame: Top-level container for creating GUI applications.

Swing Components

Swing provides GUI components like JLabel, JButton, etc.

Example:

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

public class SwingDemo {
   public static void main(String[] args) {
      JFrame frame = new JFrame("Swing Demo");
      JLabel label = new JLabel("Enter Name:");
      JTextField textField = new JTextField(15);
      JButton button = new JButton("Submit");

      button.addActionListener(e -> JOptionPane.showMessageDialog(frame, "Hello, " +
         textField.getText()));

      JPanel panel = new JPanel();
      panel.add(label);
      panel.add(textField);
      panel.add(button);
```

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

Layout Managers

Layout managers organize components in a container.

Example:

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

public class LayoutDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Layout Manager Demo");
        frame.setLayout(new BorderLayout());

        frame.add(new JButton("North"), BorderLayout.NORTH);
        frame.add(new JButton("South"), BorderLayout.SOUTH);
        frame.add(new JButton("East"), BorderLayout.EAST);
        frame.add(new JButton("West"), BorderLayout.WEST);
        frame.add(new JButton("Center"), BorderLayout.CENTER);

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

- **14.4.3 Menus**
    - **Menu Components**:
        - JMenuBar: Holds menus.
        - JMenu: Dropdown menu.
        - JMenuItem: Selectable items in a menu.
        - JSeparator: Divides menu items visually.

Menus

Swing provides classes for creating menus like JMenuBar, JMenu, etc.

Example:

```
import javax.swing.*;

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

      JMenuBar menuBar = new JMenuBar();
      JMenu fileMenu = new JMenu("File");
      JMenuItem openItem = new JMenuItem("Open");
      JMenuItem exitItem = new JMenuItem("Exit");

      exitItem.addActionListener(e -> System.exit(0));

      fileMenu.add(openItem);
      fileMenu.add(new JSeparator());
      fileMenu.add(exitItem);
      menuBar.add(fileMenu);

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

14.5 Unit Summary

This unit focuses on Applet lifecycle, its architecture, key methods, and integration of Swing components in applets.

14.6 Check Your Progress

- **Short questions**:
    1. What is the role of init() in the applet lifecycle?
    2. How can you pass parameters to an applet, and which method retrieves them?
    3. Differentiate between getCodeBase() and getDocumentBase().
    4. Write a code snippet to draw a rectangle and an oval using the Graphics class.
    5. Explain the purpose of JTextField and JTextArea.
- **Practical Exercises**:
    1. Create an applet that accepts two numbers as parameters and displays their sum.
    2. Develop a GUI application using Swing components such as JLabel, JButton, and JTextField.
    3. Design a menu bar with options for “File,” “Edit,” and “Help” using JMenu components.

Multiple-Choice Questions (MCQs)

1. Which method is used to retrieve the URL of the directory containing the applet code? a) getDocumentBase()
   b) getCodeBase()
   c) showDocument()
   d) getAppletInfo()
   Answer: b) getCodeBase()
2. In Swing, which layout manager arranges components in a row, either horizontally or vertically?
   a) BorderLayout
   b) FlowLayout
   c) GridLayout
   d) CardLayout
   Answer: b) FlowLayout
3. What is the return type of the getDocumentBase() method?
   a) String
   b) URL
   c) URI
   d) File
   Answer: b) URL
4. Which method of the Graphics class is used to draw an oval?
   a) drawPolygon()
   b) drawOval()
   c) drawRectangle()
   d) fillOval()
   Answer: b) drawOval()
5. Which interface in applet programming is used to control the playback of audio clips?
   a) AudioStream
   b) AudioClip
   c) SoundInterface
   d) AppletContext
   Answer: b) AudioClip
6. In Swing, which component is used to create a scrollable area?
   a) JScrollPane
   b) JScrollBar
   c) JPanel
   d) JFrame
   Answer: a) JScrollPane
7. What is the purpose of the AppletContext interface?
   a) To draw graphics
   b) To retrieve applet-related information
   c) To provide access to other applets on the same page
   d) To manage the applet lifecycle
   Answer: c) To provide access to other applets on the same page

8. Which of these components allows the selection of multiple items?
   a) JComboBox
   b) JList
   c) JCheckBox
   d) JRadioButton
   Answer: b) JList
9. In the applet lifecycle, which method is called only once during initialization?
   a) start()
   b) stop()
   c) init()
   d) destroy()
   Answer: c) init()
10. What method is used to handle mouse events in an anonymous inner class?
    a) mousePressed()
    b) addMouseListener()
    c) mouseClicked()
    d) mouseEvent()
    Answer: b) addMouseListener()

True/False Questions

1. The getCodeBase() method returns the URL of the document containing the applet.
   False
2. The drawLine() method of the Graphics class requires four parameters: x1, y1, x2, and y2.
   True
3. The JPanel component in Swing cannot have other components added to it.
   False
4. The JRadioButton component allows multiple selections at once.
   False
5. JMenuBar is used to create a menu bar in Swing applications.
   True

Fill in the Blanks

1. The _____ method of the Graphics class is used to draw a rectangle.
   Answer: drawRect
2. In applet architecture, the _____ method is invoked when the applet is no longer needed.
   Answer: destroy
3. The _____ layout manager divides the container into five regions: North, South, East, West, and Center.
   Answer: BorderLayout

4. The _____ component in Swing allows users to input single-line text.
   Answer: JTextField
5. The _____ interface in applet programming provides access to other applets and resources in the same context.
   Answer: AppletContext

Module V: Java Beans, JDBC, Java Servlets (10 Hours)

Unit 15: **Java Beans**
15.1 Introduction
15.2 Introducing JavaBeans Concepts and Bean Development Kit (BDK)
15.3 Using the Bean Box, Writing a simple Bean, Bean Properties (simple properties)
15.4 Manipulating events in the Bean Box
15.5 Unit Summary
15.6 Check your progress

15.1 Introduction

Java Beans is a reusable software component model in Java, designed for creating modular and reusable building blocks for applications. Java Beans simplify the development process by encapsulating properties, methods, and events into a single object. This unit introduces Java Beans concepts, their significance, and tools used for their development and manipulation.

15.2 Bean Development Kit (BDK)

JavaBeans Concepts:

1. **Definition**: A Java Bean is a reusable, self-contained software component that can be manipulated visually in a builder tool.
2. **Characteristics**:
    - **Serializable**: Beans can save their state for later use.
    - **No-argument Constructor**: Allows easy instantiation within visual tools.
    - **Getter/Setter Methods**: Provide controlled access to the properties.

Bean Development Kit (BDK): The **BDK** is a toolset provided by Sun Microsystems (now Oracle) that allows developers to create and test JavaBeans. Key components include:

- **The Bean Box**: A testing environment to visually manipulate and connect beans.
- **Utility Classes**: For property manipulation and event handling.

Example: Creating a simple Java Bean class:

```
import java.io.Serializable;

public class SimpleBean implements Serializable {
   private String message;

   public SimpleBean() {
      message = "Hello, Java Beans!";
```

```
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

15.3 Using the Bean Box, Writing a Simple Bean, and Bean Properties

Using the Bean Box: The Bean Box is an environment provided in BDK for visually testing and connecting beans. Developers can:

- Drag and drop beans onto a workspace.
- Set properties using a property sheet.
- Link events between beans.

Steps to Write a Simple Bean:

1. Create a public class that implements Serializable.
2. Provide a no-argument constructor.
3. Define properties using private fields and expose them via public getter and setter methods.

Simple Properties: Properties are attributes of a bean that control its behavior or appearance. For example:

- **Readable Property**: Only getter is provided.
- **Writable Property**: Only setter is provided.
- **Read/Write Property**: Both getter and setter are provided.

Example:

```
public class CounterBean implements Serializable {
    private int count;

    public CounterBean() {
        count = 0;    }

    public int getCount() {
        return count;    }

    public void setCount(int count) {
```

```
        this.count = count;
    }
}
```

15.4 Manipulating Events in the Bean Box

Event Handling in JavaBeans:
Beans communicate with each other using events. Key components include:

- **Event Source**: The bean that generates an event.
- **Event Listener**: The bean that handles the event.

Steps to Handle Events in the Bean Box:

1. **Add Event Listeners**: Connect one bean's events to another's methods using the Bean Box.
2. **Implement Listener Interfaces**: Use standard Java listener interfaces (ActionListener, PropertyChangeListener, etc.) for custom behavior.
3. **Event Propagation**: Customize the bean's ability to fire events.

Example: A button bean emitting an action event to a label bean:

```
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        label.setText("Button Clicked!");
    }
});
```

15.5 Unit Summary

In this unit, you learned:

- What Java Beans are and their characteristics.
- The role of the Bean Development Kit (BDK) and the Bean Box.
- How to write and test a simple Java Bean.
- How to manipulate properties and handle events within the Bean Box.

15.6 Check Your Progress

1. What are the primary characteristics of a Java Bean?
2. Explain the role of Serializable in Java Beans.
3. Write a simple Java Bean class with a name property.

4. How are events handled between beans in the Bean Box?
5. What is the significance of a no-argument constructor in Java Beans?

Multiple Choice Questions (MCQs)

1. **What is the main purpose of JavaBeans?**
   a) To enhance the performance of Java programs
   b) To develop reusable software components
   c) To create graphical user interfaces (GUIs)
   d) To manipulate database connections

   Answer: b) To develop reusable software components

2. **Which tool is primarily used to test and develop JavaBeans?**
   a) Java Development Kit (JDK)
   b) Bean Development Kit (BDK)
   c) NetBeans IDE
   d) Eclipse IDE

   Answer: b) Bean Development Kit (BDK)

3. **What is the function of the Bean Box in the BDK?**
   a) Debugging Java applications
   b) Providing a graphical environment to test JavaBeans
   c) Managing database connections
   d) Compiling Java programs

   Answer: b) Providing a graphical environment to test JavaBeans

4. **What type of property is directly accessed using getter and setter methods?**
   a) Indexed property
   b) Simple property
   c) Constrained property
   d) Bound property

   Answer: b) Simple property

5. **Which method must a bean implement to handle events?**
   a) eventHandler()
   b) actionPerformed()
   c) processEvent()
   d) addEventListener()

   Answer: c) processEvent()

6. **What interface must a class implement to be considered a JavaBean?**
   a) Serializable
   b) Cloneable
   c) Runnable
   d) None of the above

   Answer: a) Serializable

7. **How does the Bean Box manipulate events?**
   a) By linking event listeners to event sources
   b) By creating new events automatically
   c) By removing event listeners dynamically
   d) By compiling event code

   Answer: a) By linking event listeners to event sources

8. **What is the default method to define a property in a JavaBean?**
   a) setProperty()
   b) property()
   c) Getter and setter methods
   d) init()

   Answer: c) Getter and setter methods

9. **What is an event in the context of JavaBeans?**
   a) A method
   b) A user action or system action
   c) A property of the bean
   d) A GUI element

   Answer: b) A user action or system action

10. **Which annotation is used to mark a method as a property setter in a JavaBean?**
    a) @Setter
    b) @Property
    c) None, JavaBeans rely on naming conventions
    d) @SetProperty

    Answer: c) None, JavaBeans rely on naming conventions

True or False

1. The Bean Box is used to create executable Java programs.
   False

2. JavaBeans must be serializable to support persistence.
   True
3. A simple property can have multiple values associated with it.
   False
4. The Bean Development Kit (BDK) provides tools to test and run JavaBeans.
   True
5. Event listeners in JavaBeans are used to respond to events generated by event sources.
   True

Fill in the Blanks

1. JavaBeans use _____ and _____ methods to access and modify their properties.
   Answer: getter, setter
2. The _____ is a tool provided by the BDK to test JavaBeans.
   Answer: Bean Box
3. A JavaBean must implement the _____ interface to be serializable.
   Answer: Serializable
4. The property name for a getter method getAge() would be _____.
   Answer: age
5. In JavaBeans, an _____ occurs when a user interacts with a graphical component.
   Answer: event

Unit 16: **Java database connectivity**

16.1 Introduction

16.2 Introduction to JDBC, type of JDBC connectivity

16.2 Establishing database connections

16.3 Accessing relational database from Java programs

16.5 Unit Summary
16.6 Check your progress

16.1 Introduction

In modern software development, databases are integral to applications, enabling data storage, retrieval, and manipulation. Java Database Connectivity (JDBC) is a core API in Java that provides a standardized method to interact with relational databases. JDBC abstracts database-specific implementation details, offering a consistent interface for developers.

This unit introduces the fundamental concepts of JDBC, types of connectivity, and techniques for establishing and managing database connections in Java applications.

16.2 Introduction to JDBC and Types of JDBC Connectivity

What is JDBC?

JDBC is a Java-based API that allows Java applications to interact with relational databases. It acts as a bridge between Java applications and the database, enabling seamless communication for operations like querying, updating, and deleting records.

Key Features of JDBC

- Database-independent API.
- Supports various relational databases (MySQL, PostgreSQL, Oracle, etc.).
- Offers functionalities for transaction management and batch processing.

Types of JDBC Drivers

1. **Type 1: JDBC-ODBC Bridge Driver**
    - Converts JDBC calls into ODBC calls.
    - Requires an ODBC driver.
    - Example: Useful for legacy systems but not preferred due to performance limitations.
2. **Type 2: Native-API Driver**
    - Converts JDBC calls into native database-specific calls using client-side libraries.
    - Example: Oracle's OCI driver.

3. **Type 3: Network Protocol Driver**
    - Uses a middleware server to translate JDBC requests into database-specific calls.
    - Suitable for web applications.
4. **Type 4: Thin Driver**
    - Pure Java driver communicating directly with the database using a network protocol.
    - Example: MySQL Connector/J.

16.3 Establishing Database Connections

Steps to Establish a Connection

1. **Load the JDBC Driver**
   Use the Class.forName() method to load the driver class. Example:

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

2. **Create a Connection**
   Use DriverManager to establish a connection. Example:

   ```
   Connection connection = DriverManager.getConnection
   ("jdbc:mysql://localhost:3306/mydb", "username", "password");
   ```

3. **Create a Statement or PreparedStatement**
   Statements allow executing SQL queries. Example:

   ```
   Statement stmt = connection.createStatement();
   String sql = "SELECT * FROM employees";
   ResultSet rs = stmt.executeQuery(sql);
   ```

4. **Execute the Query and Process Results**
   ```
   while (rs.next()) {

      System.out.println("Employee ID: " + rs.getInt("id"));
      System.out.println("Name: " + rs.getString("name"));
   }
   ```

5. **Close the Resources**
   Always close the ResultSet, Statement, and Connection objects to free resources.

   ```
   rs.close();
   stmt.close();
   connection.close();
   ```

16.4 Accessing Relational Databases from Java Programs

To access relational databases, follow these steps:

1. **Set Up the Database**
   Create a sample database and table. Example for MySQL:

   ```
   CREATE DATABASE mydb;
   CREATE TABLE employees (
      id INT PRIMARY KEY,
      name VARCHAR(50),
      salary DECIMAL(10,2)
   );
   ```

2. **Connect Java Program to the Database**
   ```
   import java.sql.*;

   public class DatabaseDemo {
      public static void main(String[] args) {
         String url = "jdbc:mysql://localhost:3306/mydb";
         String user = "root";
         String password = "password";

         try (Connection connection = DriverManager.getConnection(url, user, password)) {
            System.out.println("Connected to the database!");

            String query = "SELECT * FROM employees";
            Statement statement = connection.createStatement();
            ResultSet resultSet = statement.executeQuery(query);

            while (resultSet.next()) {
               System.out.println("ID: " + resultSet.getInt("id"));
               System.out.println("Name: " + resultSet.getString("name"));
               System.out.println("Salary: " + resultSet.getBigDecimal("salary"));
            }
         } catch (SQLException e) {
            e.printStackTrace();
         }
      }
   }
   ```

16.5 Unit Summary

This unit introduced JDBC as a crucial technology for Java developers working with relational databases. Key takeaways include:

- Understanding the role and features of JDBC.
- Exploring the types of JDBC drivers.
- Establishing connections and accessing databases using JDBC.
- Implementing database operations like querying and processing results in Java programs.

16.6 Check Your Progress

1. **What are the key features of JDBC?**
2. **Explain the differences between Type 1 and Type 4 JDBC drivers.**
3. **Write the Java code to connect to a MySQL database and retrieve all rows from a table named products.**
4. **Why is it important to close the database connection in Java programs?**
5. **List the steps to execute a SQL query using JDBC.**

Multiple Choice Questions (MCQs)

1. **What does JDBC stand for?**
   a) Java Database Communication
   b) Java Database Connectivity
   c) Java Dynamic Connection
   d) Java Data Compilation
   Answer: b) Java Database Connectivity
2. **Which JDBC driver type is also known as the "thin driver"?**
   a) Type-1
   b) Type-2
   c) Type-3
   d) Type-4
   Answer: d) Type-4
3. **Which JDBC API class is used to establish a database connection?**
   a) DriverManager
   b) Connection
   c) Statement
   d) ResultSet
   Answer: a) DriverManager
4. **What is the first step to accessing a database in JDBC?**
   a) Writing a SQL query
   b) Loading the database driver
   c) Creating a ResultSet
   d) Executing an update query
   Answer: b) Loading the database driver
5. **Which method is used to execute a query in a Statement object?**
   a) executeUpdate()
   b) executeQuery()
   c) execute()

d) All of the above
Answer: d) All of the above

6. **What does the executeUpdate method return?**
   a) A ResultSet object
   b) Number of rows affected
   c) A Boolean value
   d) None of the above
   Answer: b) Number of rows affected
7. **Which type of JDBC driver is a bridge to other database connectivity APIs?**
   a) Type-1
   b) Type-2
   c) Type-3
   d) Type-4
   Answer: a) Type-1
8. **Which object is used to retrieve query results in JDBC?**
   a) Statement
   b) ResultSet
   c) Connection
   d) DriverManager
   Answer: b) ResultSet
9. **Which type of JDBC driver uses native library calls for database communication?**
   a) Type-1
   b) Type-2
   c) Type-3
   d) Type-4
   Answer: b) Type-2
10. **What is the purpose of the close() method in JDBC?**
    a) To end the SQL query execution
    b) To close the connection to the database
    c) To finalize the driver manager
    d) None of the above
    Answer: b) To close the connection to the database

True or False

1. JDBC is used for Java programs to interact with databases.
   Answer: True
2. Type-3 JDBC drivers use database-specific APIs for connectivity.
   Answer: False
3. A Connection object represents a session between a Java application and a database.
   Answer: True
4. The Statement interface can be used to execute parameterized SQL queries.
   Answer: False
5. The Type-4 JDBC driver is platform-dependent.
   Answer: False

Fill in the Blanks

1. The JDBC API is part of the _____ package in Java.
   Answer: java.sql
2. The _____ interface in JDBC is used to execute SQL queries.
   Answer: Statement
3. To connect to a database, a valid _____ URL is required.
   Answer: JDBC
4. The _____ method is used to load a driver class in JDBC.
   Answer: Class.forName
5. In JDBC, a _____ object is used to traverse and manipulate the results of a query.
   Answer: ResultSet