ਜਗਤ ਗੁਰੂ ਨਾਨਕ ਦੇਵ
ਪੰਜਾਬ ਸਟੇਟ ਓਪਨ ਯੂਨੀਵਰਸਿਟੀ
ਪਟਿਆਲਾ

# JAGAT GURU NANAK DEV

# PUNJAB STATE OPEN UNIVERSITY, PATIALA

**(Established by Act No. 19 of 2019 of the Legislature of State of Punjab)**

**The Motto of the University**

**(SEWA)**

**SKILL ENHANCEMENT EMPLOYABILITY WISDOM ACCESSIBILITY**


## M.Sc. (Computer Science)

## Course Name: Web Programming

## Course Code: MSCS-3-01T

**ADDRESS: C/28, THE LOWER MALL, PATIALA-147001**

**WEBSITE: www.psou.ac.in**

# JAGAT GURU NANAK DEV
# PUNJAB STATE OPEN UNIVERSITY PATIALA
**(Established by Act No.19 of 2019 of Legislature of the State of Punjab)**

## Faculty of School of Science and Emerging Technologies:

**Dr. Baljit Singh Khera (Head)**
Professor, School of Sciences and Emerging Technologies
Jagat Guru Nanak Dev Punjab State Open University, Patiala

**Dr. Kanwalvir Singh Dhindsa**
Professor, School of Sciences and Emerging Technologies
Jagat Guru Nanak Dev Punjab State Open University, Patiala

**Dr. Amitoj Singh**
Associate Professor, School of Sciences and Emerging Technologies
Jagat Guru Nanak Dev Punjab State Open University, Patiala

**Dr. Karan Sukhija**
Assistant Professor, School of Sciences and Emerging Technologies
Jagat Guru Nanak Dev Punjab State Open University, Patiala

**Dr. Monika Pathak**
Assistant Professor, School of Sciences and Emerging Technologies
Jagat Guru Nanak Dev Punjab State Open University, Patiala

**Dr. Gaurav Dhiman**
Assistant Professor, School of Sciences and Emerging Technologies
Jagat Guru Nanak Dev Punjab State Open University, Patiala

## Faculty of School of Business Management & Commerce:

**Dr. Pooja Aggarwal**
Assistant Professor, School of Business & Commerce
Jagat Guru Nanak Dev Punjab State Open University, Patiala

## Faculty of School of Social Sciences and Liberal Arts:

**Dr. Pinky Sra**
Assistant Professor, School of Social Sciences and Liberal Arts
Jagat Guru Nanak Dev Punjab State Open University, Patiala

**JAGAT GURU NANAK DEV**
**PUNJAB STATE OPEN UNIVERSITY PATIALA**
**(Established by Act No.19 of 2019 of Legislature of the State of Punjab)**

**PROGRAMME COORDINATOR**
Dr. Karan Sukhija (Assistant Professor)
School of Sciences and Emerging Technologies
JGND PSOU, Patiala

**COURSE COORDINATOR AND EDITOR:**
Dr. Gaurav Dhiman (Assistant Professor)
School of Sciences and Emerging Technologies
JGND PSOU, Patiala

**COURSE OUTCOMES:**

- Demonstrate mastery in Java programming for web development.
- Familiarity with Java programming for efficient and modular web application development.
- Ability to integrate Java applications with databases using Java Database Connectivity (JDBC) for effective data management.
- Design and implement of web services in Java
- Understand and implement security measures specific to Java web development, including authentication, authorization, and protection against common vulnerabilities.

# JAGAT GURU NANAK DEV
# PUNJAB STATE OPEN UNIVERSITY PATIALA
**(Established by Act No.19 of 2019 of Legislature of the State of Punjab)**

## PREFACE

Jagat Guru Nanak Dev Punjab State Open University, Patiala was established in Decembas 2019 by Act 19 of the Legislature of State of Punjab. It is the first and only Open Universit of the State, entrusted with the responsibility of making higher education accessible to all especially to those sections of society who do not have the means, time or opportunity to pursue regular education.

In keeping with the nature of an Open University, this University provides a flexible education system to suit every need. The time given to complete a programme is double the duration of a regular mode programme. Well-designed study material has been prepared in consultation with experts in their respective fields.

The University offers programmes which have been designed to provide relevant, skill-based and employability-enhancing education. The study material provided in this booklet is self instructional, with self-assessment exercises, and recommendations for further readings. The syllabus has been divided in sections, and provided as units for simplification.

The Learner Support Centres/Study Centres are located in the Government and Government aided colleges of Punjab, to enable students to make use of reading facilities, and for curriculum-based counselling and practicals. We, at the University, welcome you to be a part of this institution of knowledge.

Prof. G. S. Batra,
Dean Academic Affairs

# M.Sc. (Computer Science)
# Semester-3 (Syllabus)
# MSCS-3-01T: Web Programming

**Total Marks: 100**
**External Marks: 70**
**Internal Marks: 30**
**Credits: 4**
**Pass Percentage: 40%**

**SECTION-A**

**Unit 1: Java and the Internet:** The Java programming language and its characteristics; Java development kit, Java run- time environment; Java compiler.

**Unit II: Fundamentals of Java:** Java Vs. C++, Byte Code, Java Virtual Machine, constants, variables, data types, operators, expressions, control structures, defining class, creating objects, accessing class members, constructors, Garbage Collection, method overloading.

**Unit III: Inheritance:** Different types of Inheritance, member access, using super keyword to call super class constructors, creating a multilevel hierarchy, method overriding, dynamic method dispatch, using abstract classes, using Final keyword.

**Unit IV: I/O Basics:** streams, the predefined streams; Reading console Input, Writing console Output. Arrays and Strings: One-dimensional and two-dimensional Arrays, String Handling using String and String Buffer class, String Functions.

**SECTION-B**

**Unit V: Packages:** Types of packages, defining a package, importing packages, Access protection Interfaces: Defining an Interface, Implementing Interfaces, Variables in Interfaces, achieving multiple inheritance using interfaces, Interface and Abstract classes.

**Unit VI: Exception Handling:** Java Exception handling model, Types of exception, using Try and catch, Multiple Try and Catch clauses, Nested Try statements, finally block, user defined exceptions.

**Unit VII: Multi-threaded Programming:** The Java Thread model, the Thread class and Runnable interface, creating a Thread using Runnable Interface and extending Thread, Creating Multiple Threads, Thread Priorities, Synchronizations: Methods, Statements, Inter Thread Communication, Deadlock, Suspending, Resuming and Stopping Threads.

**Unit VIII: Applet Programming:** Introduction, Types of applet, Life Cycle, incorporating an applet into web page using Applet Tag, running applets, using Graphics class and its methods to draw lines, rectangles, circles, ellipses, arcs and polygons

# M.Sc. (Computer Science)
# SEMESTER-3
# COURSE: Web Programming

| UNIT 1: JAVA AND THE INTERNET |
|---|
| 1.1 The Java programming language and its characteristics |
| 1.2 Java development kit |
| 1.3 Java run- time environment |
| 1.4 Java compiler |

## 1.1 The Java Programming Language and Its Characteristics

The Java programming language, initially developed by Sun Microsystems in the mid-1990s, has emerged as one of the most widely used and influential programming languages in the world. Known for its platform independence, robustness, and versatility, Java has become a staple in various domains, from web development to enterprise applications, mobile development, and embedded systems. This article provides an in-depth exploration of the Java programming language, its key characteristics, and its impact on the software development landscape.

### Historical Overview

Java's journey began in 1991 when a team at Sun Microsystems, led by James Gosling, started developing a language for programming household appliances. However, the project evolved, and by 1995, Java had transformed into a general-purpose programming language. One of its defining features was the "Write Once, Run Anywhere" (WORA) principle, emphasizing its cross-platform capabilities.

### Key Characteristics of Java

**1. Platform Independence**

Java's platform independence is a fundamental characteristic that sets it apart. Java code is compiled into an intermediate form called bytecode, which can run on any device equipped with a Java Virtual Machine (JVM). This "write once, run anywhere" capability has been a key driver of Java's widespread adoption.

**2. Object-Oriented Programming**

Java is an object-oriented programming (OOP) language, meaning it revolves around the concept of objects. Encapsulation, inheritance, and polymorphism are core principles of Java's OOP paradigm. This approach promotes modular and reusable code, enhancing software maintainability and scalability.

**3. Robust and Secure**

Java's design includes features aimed at ensuring robustness and security. Memory management is handled by the Java Virtual Machine, which includes garbage collection to automatically reclaim unused memory. Additionally, Java enforces strong type checking during compilation, reducing the likelihood of runtime errors.

**4. Multithreading**

Java provides built-in support for multithreading, allowing developers to write concurrent, scalable, and responsive applications. Multithreading is crucial for tasks such as handling user interfaces,

background processing, and managing concurrent connections in networked applications.

**5. Distributed Computing**
Java's architecture supports distributed computing through its Remote Method Invocation (RMI) mechanism and Java's extensive set of APIs. This makes it well-suited for developing networked and distributed applications, a key requirement in the modern computing landscape.

**6. Dynamic and Extensible**
Java's dynamic nature allows for dynamic loading of classes, enabling applications to adapt and extend their functionality at runtime. This extensibility is particularly valuable in scenarios where new functionality needs to be added without restarting the entire application.

**7. Rich Standard Library**
Java comes with a comprehensive standard library that provides a wide array of classes and packages for common programming tasks. This includes utilities for networking, file I/O, data structures, and more. The rich standard library accelerates development by offering pre-built components and reducing the need for developers to reinvent the wheel.

**8. Community Support and Ecosystem**
Java benefits from a vibrant and expansive community of developers. The Java Community Process (JCP) facilitates collaboration and the evolution of the language through the creation of Java Specification Requests (JSRs). Additionally, the Java ecosystem includes a plethora of frameworks, tools, and third-party libraries that enhance productivity and enable developers to tackle a variety of challenges.

**9. Versatility: From Desktop to Mobile to the Cloud**
Java's versatility is evident in its ability to power a diverse range of applications. From desktop applications using JavaFX to mobile applications on the Android platform to cloud-based microservices and server-side applications, Java's adaptability makes it suitable for a wide spectrum of use cases.

## Java Language Components

**1. Java Virtual Machine (JVM)**
At the heart of Java's platform independence is the JVM. It interprets and executes Java bytecode, serving as an abstraction layer between the compiled Java code and the underlying hardware. Different implementations of the JVM exist, catering to various platforms.

**2. Java Development Kit (JDK) and Java Runtime Environment (JRE)**
The JDK includes the tools needed for Java development, such as the compiler and debugger, along with the JRE. The JRE, on the other hand, contains the JVM and libraries necessary for running Java applications. For end-users, the JRE is sufficient, while developers require the full JDK for coding and compiling.

**3. Java Standard Edition (SE) and Java Enterprise Edition (EE)**
Java SE is the standard edition, encompassing the core features of the language. Java EE, now known as Jakarta EE, extends Java SE to provide additional APIs for enterprise-level applications. These include features for web services, messaging, and persistence.

**4. JavaFX**
JavaFX is a platform for creating rich internet applications (RIAs). It provides a set of APIs for creating graphical user interfaces (GUIs) and is used for developing desktop and mobile applications. JavaFX supports multimedia, 2D and 3D graphics, and has built-in support for web standards.

## Java Development Process

### 1. Writing Java Code
The development process typically begins with writing Java code using an integrated development environment (IDE) or a text editor. Java code is saved in files with a `.java` extension.

### 2. Compilation
The Java compiler (`javac`) translates the human-readable Java code into bytecode. This bytecode is a set of instructions for the JVM and is stored in files with a `.class` extension.

### 3. Bytecode Execution
The JVM executes the bytecode. This step is where the platform independence of Java becomes evident. As long as a device has a compatible JVM, it can run Java bytecode, regardless of the underlying hardware and operating system.

### 4. Garbage Collection
Java's automatic garbage collection mechanism manages memory by identifying and reclaiming unused objects. This feature alleviates developers from manual memory management concerns, reducing the risk of memory leaks.

## Java in Modern Development

### 1. Mobile Development with Android
Java has played a pivotal role in mobile app development, particularly on the Android platform. Android applications are primarily developed using Java (or Kotlin). The portability of Java code has enabled developers to create applications that run on a myriad of Android devices seamlessly.

### 2. Web Development with Spring Boot
Java's presence in web development has been bolstered by frameworks like Spring and Spring Boot. Spring Boot simplifies the development of production-ready applications, providing a convention-over-configuration approach and a range of features for building robust and scalable web services.

### 3. Microservices Architecture
Java has adapted well to the microservices architecture, where applications are developed as a collection of loosely coupled services. Frameworks like Spring Cloud facilitate the creation and management of distributed systems, supporting developers in building scalable and resilient microservices.

### 4. Cloud-Native Development
As organizations embrace cloud computing, Java has evolved to meet the demands of cloud-native development. Containers and orchestration tools like Docker and Kubernetes are well-supported in the Java ecosystem, enabling seamless deployment and scaling of Java applications in cloud environments.

## Challenges and Future Trends

### 1. Competition from New Languages
While Java remains a powerhouse in the software development landscape, it faces competition from newer languages like Kotlin, which is now the preferred language for Android development, and languages like Rust and Swift, which are gaining traction in specific domains.

### 2. Security Concerns
Despite its emphasis on security, Java has faced challenges, including security vulnerabilities. Ongoing efforts within the community and the adoption of best practices help mitigate these concerns,

but security remains a key area of focus.

**3. Containerization and Serverless Computing**
Java continues to adapt to emerging trends in containerization and serverless computing. The lightweight and modular features introduced in recent Java versions aim to optimize Java applications for these modern deployment models.

**4. Project Loom and Fiber**
Project Loom, an ongoing effort in the Java community, aims to simplify concurrent programming using fibers, lightweight threads that can be used to write highly concurrent applications more efficiently. This project represents a significant step toward enhancing Java's capabilities in the era of multi-core processors.

**Conclusion**

The Java programming language has withstood the test of time and remains a cornerstone in the world of software development. Its platform independence, robustness, and adaptability have contributed to its enduring popularity. As technology continues to evolve, Java's ability to embrace new trends and address emerging challenges will determine its continued relevance in the dynamic landscape of programming languages. The Java community, with its commitment to innovation and collaboration, plays a crucial role in shaping the language's future. Whether in mobile development, web services, microservices architecture, or cloud-native applications, Java continues to be a versatile and powerful choice for developers worldwide.

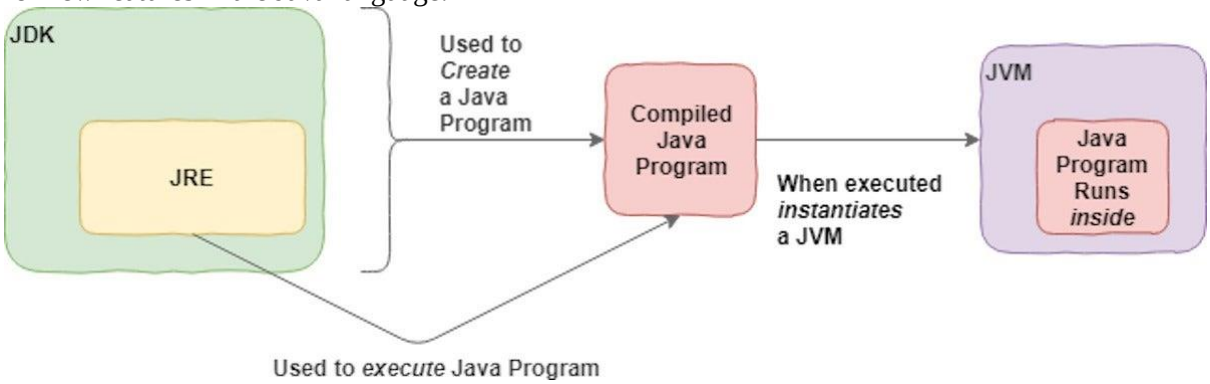
# 1.2 Java Development Kit (JDK): Unveiling the Toolkit for Java Developers

**Introduction**

The Java Development Kit (JDK) is a comprehensive software development kit that provides the necessary tools, executables, and binaries for Java application development. Developed by Oracle Corporation, the JDK facilitates the entire software development life cycle, from writing and compiling code to debugging and testing. In this exploration, we delve into the components, features, and significance of the Java Development Kit.

**Evolution of the JDK**

The history of the JDK is intertwined with the evolution of the Java programming language. The first version of Java, released in 1995, included a basic set of tools for development. As Java gained popularity and underwent enhancements, the need for a more robust and feature-rich toolkit became evident. Subsequent versions of the JDK have introduced improvements, additional tools, and support for new features in the Java language.



## Components of the JDK

**1. Java Compiler (`javac`)**
The Java compiler is a fundamental component of the JDK, responsible for translating human-readable Java source code into bytecode. This intermediate bytecode is platform-independent and can be executed by the Java Virtual Machine (JVM). The compiler ensures that the code adheres to the syntax and rules of the Java programming language.

**2. Java Virtual Machine (JVM)**
At the heart of Java's platform independence is the JVM. It interprets and executes the compiled Java bytecode, providing a runtime environment that abstracts the underlying hardware and operating system. Different implementations of the JVM exist for various platforms, ensuring that Java applications can run consistently across diverse environments.

**3. Java Runtime Environment (JRE)**
The JRE is a subset of the JDK and includes the JVM along with essential libraries and components required to run Java applications. For end-users who only need to execute Java applications, the JRE is sufficient. However, developers require the full JDK to compile and build applications.

**4. Java Development Kit (JDK) Tools**

The JDK includes a rich set of tools that streamline the development process. Some key tools are:

- Java Archive (JAR) Tool (`jar`): This tool is used to package Java applications and libraries into JAR files. JAR files are a standard way to bundle and distribute Java code.

- Java Documentation Generator (`javadoc`): Developers can use this tool to automatically generate documentation from their source code comments. The resulting documentation is in HTML format and serves as a reference for other developers using the code.

- Java Debugger (`jdb`): The debugger allows developers to identify and fix issues in their code by providing features for setting breakpoints, inspecting variables, and stepping through code execution.

- Java Applet Viewer (`appletviewer`): While applets are less prevalent today, this tool was historically used to test and run Java applets outside of a web browser.

- JavaFX Packager: With the inclusion of JavaFX in the JDK, tools for packaging JavaFX applications for deployment are also provided.

**5. JavaFX**
JavaFX, included as part of the JDK, is a platform for creating rich internet applications (RIAs). It provides a set of APIs for building interactive and visually appealing user interfaces. JavaFX supports features such as 2D and 3D graphics, multimedia, and integration with web standards. The inclusion of JavaFX in the JDK highlights Oracle's commitment to providing a comprehensive toolkit for modern Java development.

**6. Java Mission Control and Java Flight Recorder**
Java Mission Control (JMC) and Java Flight Recorder (JFR) are tools for monitoring, managing, and profiling Java applications. They provide insights into application performance, resource consumption, and other runtime characteristics. JFR, in particular, allows developers to record and analyze detailed information about the application's behavior, helping identify performance bottlenecks and optimize code.

## Installing and Configuring the JDK

**1. Download and Installation**
To use the JDK, developers need to download and install it on their development machines. Oracle, the primary maintainer of the JDK, provides official distributions for various operating systems. Additionally, alternative distributions such as OpenJDK, AdoptOpenJDK, and Amazon Corretto are available, offering open-source implementations of the JDK.

**2. Environment Variables**
After installation, configuring the system's environment variables is essential to ensure that the operating system recognizes the JDK's executable files. Key environment variables include `JAVA_HOME` (pointing to the JDK installation directory) and `PATH` (including the `bin` directory of the JDK). Proper configuration enables developers to run Java commands and tools from the command line or scripts.

## The significance of the JDK in Java Development

**1. Complete Development Environment**
The JDK provides a complete and integrated development environment for Java developers. With tools for compiling, debugging, documenting, and packaging Java applications, developers can perform all essential tasks without relying on external tools or third-party solutions.

**2. Cross-Platform Development**
One of Java's core principles is "Write Once, Run Anywhere" (WORA). The JDK, with its ability to generate platform-independent bytecode, contributes significantly to achieving this principle. Developers can write Java code on one platform and confidently expect it to run on any other platform with a compatible JVM.

**3. Community and Industry Standard**
The JDK is not only a product of Oracle but is also an industry standard for Java development. Its features and tools are widely adopted, and its compatibility with various IDEs (Integrated Development Environments) and build systems make it a preferred choice for Java developers across the globe. The open-source nature of some JDK distributions, such as OpenJDK, enhances transparency and collaboration within the Java community.

**4. Support for Modern Java Features**
As Java evolves, the JDK evolves with it. New versions of the JDK bring support for the latest features introduced in the Java language. Features such as lambdas, modules, and records, introduced in recent Java versions, are made accessible to developers through updated JDK releases.

**5. Security and Updates**
Oracle and other JDK maintainers actively address security vulnerabilities and provide timely updates. Developers can benefit from the latest security patches and enhancements by regularly updating their JDK installations. The commitment to security underscores the JDK's role in supporting robust and secure Java applications.

## Challenges and Future Directions

**1. Licensing Changes**
Oracle's decision to change the licensing and distribution model for Oracle JDK has led to increased attention on alternative JDK distributions, such as OpenJDK and AdoptOpenJDK. Developers need to consider licensing implications and choose a distribution that aligns with their project's requirements.

**2. Project Loom and the Future of Concurrency**
Project Loom, an initiative within the Java community, aims to simplify concurrent programming in

Java by introducing lightweight, user-mode threads called fibers. This project could significantly impact how developers approach concurrency in Java, potentially making it more accessible and efficient.

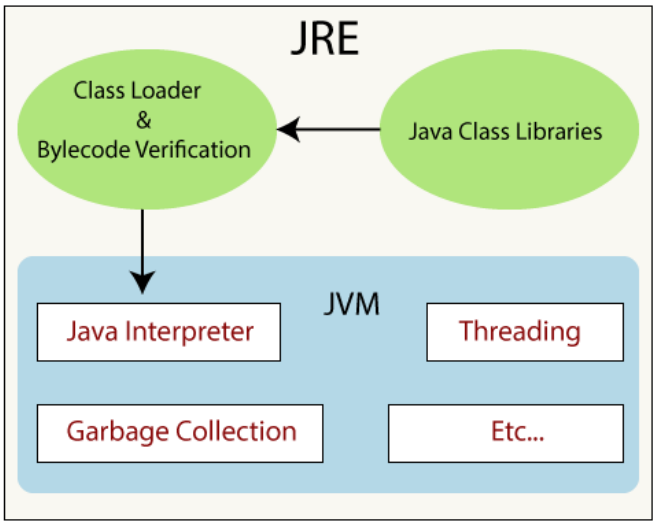
**3. Adoption of New Java Versions**
Encouraging developers to adopt the latest Java versions remains a challenge. While the introduction of a six-month release cadence for Java has accelerated the delivery of new features, some organizations and projects may be hesitant to upgrade due to concerns about compatibility and the need for thorough testing.

**Conclusion**
The Java Development Kit stands as an indispensable toolkit for Java developers, empowering them to create robust, cross-platform applications.

# 1.3 Java Runtime Environment (JRE): Unveiling the Engine of Java Applications

**Introduction**
The Java Runtime Environment (JRE) is a crucial component in the Java ecosystem, serving as the engine that enables the execution of Java applications. Developed by Oracle Corporation, the JRE provides the necessary runtime support for Java programs, allowing them to run on diverse hardware and operating systems. In this exploration, we delve into the components, functionalities, and significance of the Java Runtime Environment.

**Evolution of the JRE**
The concept of a runtime environment in the context of programming languages has evolved over the years. In the case of Java, the need for a runtime environment became evident as the language aimed to achieve platform independence through the "Write Once, Run Anywhere" (WORA) principle. The JRE, as a dedicated runtime environment for Java, has undergone refinement and expansion to accommodate new features, enhance performance, and address the evolving needs of the Java programming language.




**Components of the JRE**
**1. Java Virtual Machine (JVM)**
At the core of the JRE is the Java Virtual Machine (JVM). The JVM is responsible for interpreting and executing Java bytecode, the intermediate form of Java code generated by the Java compiler. It provides a runtime environment that abstracts the underlying hardware and operating system, enabling

Java applications to run consistently across different platforms.
- Class Loader: The class loader is responsible for loading classes into the JVM as they are referenced by the running Java program. It dynamically loads classes as needed, facilitating a more efficient and modular approach to application development.

- Bytecode Verifier: Before executing bytecode, the JVM employs a bytecode verifier to ensure that the code adheres to the rules and restrictions imposed by the Java language. This verification step enhances security by preventing the execution of malformed or malicious bytecode.

- Interpreter and Just-In-Time (JIT) Compiler: The JVM includes an interpreter that directly executes bytecode. Additionally, modern JVMs often employ a Just-In-Time compiler that translates bytecode into native machine code for improved performance. This combination of interpretation and compilation contributes to the JVM's efficiency.

- Garbage Collector: Memory management is a critical aspect of Java application development. The garbage collector within the JVM automatically identifies and deallocates memory occupied by objects that are no longer reachable, reducing the risk of memory leaks and simplifying memory management for developers.

**2. Java Application Programming Interface (API)**
The JRE includes a rich set of libraries and APIs that provide essential functionality for Java applications. These libraries cover a wide range of areas, including input/output operations, networking, graphical user interfaces, data structures, and more. The Java API simplifies development by offering pre-built components that developers can leverage, saving time and effort.

**3. Java Runtime Libraries**
The Java Runtime Libraries, a subset of the Java API, include core packages such as `java.lang`, `java.util`, and `java.io`. These libraries provide fundamental classes and utilities that form the backbone of Java applications. For example, the `java.lang` package contains essential classes like `Object`, `String`, and `System`, while `java.util` includes classes for data structures like lists, maps, and queues.

**4. Java Native Interface (JNI)**
The Java Native Interface allows Java code to interact with applications and libraries written in other languages, such as C and C++. JNI enables the incorporation of native code into Java applications when necessary. This capability is particularly useful for tasks that require low-level system access or integration with existing non-Java codebases.

## JRE Installation and Configuration

**1. Downloading and Installing the JRE**
End-users and those focused solely on running Java applications typically install the Java Runtime Environment. The installation process involves downloading the appropriate JRE distribution for the target operating system from the official Oracle website or other trusted sources. Alternatively, users can opt for open-source implementations of the JRE, such as OpenJDK or AdoptOpenJDK.

**2. Environment Variables and Configuration**
Once installed, configuring the system's environment variables is essential for proper functioning. The `PATH` variable should include the directory containing the JRE binaries to enable users to run Java applications from the command line or scripts. Additionally, setting the `JAVA_HOME` variable to point to the JRE installation directory ensures compatibility with various tools and applications that rely on the presence of the JRE.

## The Significance of the JRE in Java Development

**1. Platform Independence**
A central tenet of Java's design philosophy is its platform independence, and the JRE plays a pivotal role in realizing this goal. By providing a consistent runtime environment through the JVM, the JRE enables Java applications to execute on any device or operating system equipped with a compatible JRE implementation. This "Write Once, Run Anywhere" capability is a cornerstone of Java's success in diverse application domains.

**2. Ease of Deployment**
Java applications, packaged as bytecode and accompanied by necessary resources, can be distributed as standalone JAR (Java Archive) files. End-users only need to have a compatible JRE installed on their systems to run these applications, eliminating the need for recompilation or modification based on the target platform. This ease of deployment has contributed to the widespread adoption of Java in various contexts, from desktop applications to web services.

**3. Security and Sandboxing**
The JRE incorporates security features that contribute to the overall robustness of Java applications. The JVM's bytecode verifier ensures that only valid and safe code is executed, reducing the risk of security vulnerabilities. Additionally, the ability to run Java applications in a sandboxed environment enhances security by restricting access to certain resources and operations, mitigating potential threats.

**4. Compatibility and Interoperability**
The JRE's adherence to Java standards ensures a high degree of compatibility across different implementations. Applications developed and tested on one JRE implementation can typically run on others without modification, fostering interoperability and allowing users to choose from various JRE distributions based on their preferences or requirements.

**5. Dynamic Class Loading and Code Extensibility**
Java applications can dynamically load classes at runtime, a capability facilitated by the JRE's class-loading mechanism. This feature allows applications to adapt, extend, and modify their behavior without requiring a restart. It is particularly valuable in scenarios where plugins or modules need to be added or updated without disrupting the application's operation.

## Challenges and Future Directions

**1. Security Concerns and Updates**
While Java's security features are robust, the evolving threat landscape necessitates continuous vigilance and updates. Regular updates to the JRE address security vulnerabilities and introduce improvements. However, ensuring that end-users promptly apply these updates remains a challenge, and organizations must actively manage the security of their Java runtime environments.

**2. Containerization and Cloud-Native Development**
As modern software development embraces containerization and cloud-native architectures, adapting the JRE to these paradigms becomes crucial. Efforts to optimize the JRE for containerized environments, where lightweight and efficient runtimes are favored, are ongoing. The integration of the JRE with container orchestration tools such as Docker and Kubernetes is a key area of exploration.

**3. Project Loom and Fiber**
Project Loom, an initiative within the Java community, aims to simplify concurrent programming by introducing lightweight, user-mode threads called fibers. The success and adoption of Project Loom could reshape how Java applications handle concurrency, offering developers.

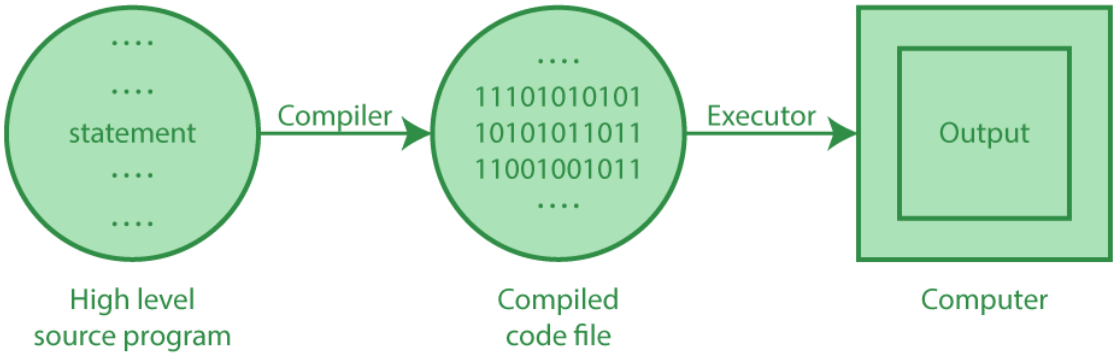
# 1.4 Java Compiler: The Architect of Bytecode Transformation

**Introduction**

The Java Compiler is a critical component of the Java development toolkit, responsible for translating human-readable Java source code into an intermediate form known as bytecode. Developed as part of the Java Development Kit (JDK) by Sun Microsystems and now maintained by Oracle, the Java Compiler plays a pivotal role in realizing the "Write Once, Run Anywhere" (WORA) principle, a core tenet of the Java programming language. In this exploration, we delve into the intricacies of the Java Compiler, examining its components, functionalities, and its significance in the software development lifecycle.

**Historical Evolution**

The evolution of the Java Compiler is closely tied to the inception and growth of the Java programming language. The first version of Java, released in 1995, featured a simple compiler that translated Java source code into bytecode. As Java gained popularity and underwent various enhancements, the compiler evolved to support new language features, optimizations, and compatibility with different platforms.




**Components of the Java Compiler**

**1. Tokenizer and Lexer**

The Java Compiler begins its journey by processing the source code through a series of lexical analysis phases. The source code is broken down into tokens, which are the smallest units of meaning in the Java language. The tokenizer and lexer work in tandem to recognize keywords, identifiers, literals, and other language elements, preparing the code for subsequent parsing.

**2. Parser**

The parser is responsible for syntactic analysis, interpreting the structure of the source code based on the grammar rules of the Java language. It constructs an abstract syntax tree (AST), representing the hierarchical structure of the code. The AST serves as an intermediate representation that captures the relationships and dependencies among different elements in the code.

**3. Semantic Analyzer**

After syntactic analysis, the compiler engages in semantic analysis to ensure that the code adheres to the rules and semantics of the Java language. This phase involves type checking, scope resolution, and the enforcement of various language constraints. The semantic analyzer plays a crucial role in identifying and reporting errors related to type mismatches, undeclared variables, and other semantic issues.

**4. Intermediate Code Generator**

With the AST and semantic analysis in place, the compiler proceeds to generate intermediate code. In the case of Java, this intermediate code is known as bytecode. Bytecode is a platform-independent

representation of the source code and serves as an intermediary between the high-level Java code and the machine code of the target platform.

**5. Optimizations**
Modern Java compilers incorporate a range of optimization techniques to enhance the performance of the generated bytecode. These optimizations may include constant folding, dead code elimination, inlining, and other transformations aimed at producing more efficient and streamlined bytecode. The goal is to improve the runtime performance of Java applications without altering their observable behavior.

**6. Code Generation**

The final step of the compilation process involves the generation of machine-specific code or bytecode. In the context of Java, the output is bytecode in the form of `.class` files. These files contain the compiled code that can be executed by the Java Virtual Machine (JVM). The code generation phase takes into account the target platform's architecture and instruction set, ensuring compatibility and optimal performance.

## Java Compiler Versions

**1. Classic Compiler (javac)**
The original Java Compiler, known as the "Classic Compiler," was the first compiler developed for the Java language. It underwent improvements and optimizations over the years, supporting features introduced in various Java versions. The Classic Compiler is still in use today and serves as a robust and reliable tool for compiling Java source code.

**2. Java Compiler API**
In addition to the traditional command-line compiler (javac), Java provides a Compiler API that allows developers to programmatically invoke the compilation process. This API, part of the javax.tools package, provides a programmatic interface for interacting with the Java Compiler. Developers can use it to dynamically compile Java code within their applications, facilitating scenarios such as runtime code generation and compilation.

**3. Ahead-of-Time (AOT) Compilation**
Java 9 introduced the concept of Ahead-of-Time (AOT) compilation with the GraalVM compiler. AOT compilation allows Java applications to be compiled into native machine code ahead of execution, potentially improving startup time and reducing the runtime footprint. While AOT compilation is not the default mode for Java applications, it provides an alternative for specific use cases and deployment scenarios.

## Java Compiler Options and Flags

The Java Compiler offers a variety of options and flags that developers can use to customize the compilation process. Some common options include:

- -classpath or -cp: Specifies the location of the class files and libraries needed during compilation.

- -d: Specifies the destination directory for the compiled bytecode.

- -source and -target: Specifies the version of the Java source code and the target bytecode version, allowing developers to maintain compatibility with specific Java versions.

- -Xlint: Enables or disables specific warnings issued by the compiler. It allows developers to control the level of detail in the compiler's feedback.

- -g: Includes debugging information in the compiled bytecode, facilitating debugging with tools like a debugger or profiler.

## Java Compiler in the Software Development Lifecycle

**1. Development Phase**
During the development phase, the Java Compiler is a fundamental tool used by developers to transform human-readable source code into executable bytecode. Developers write Java code using an integrated development environment (IDE) or a text editor, and they invoke the compiler to check for syntax errors, perform type checking, and generate bytecode.

**2. Build Automation**
In build automation systems such as Apache Maven or Gradle, the Java Compiler is a key component in the build process. These systems use the compiler to compile the source code, manage dependencies, and create the final artifacts, such as JAR files. The build process ensures that the entire codebase is compiled consistently and that dependencies are resolved.

**3. Continuous Integration and Deployment (CI/CD)**
In CI/CD pipelines, the Java Compiler is integrated to automate the compilation and testing of code changes. Automated builds triggered by version control events use the compiler to ensure that new code adheres to quality standards and does not introduce compilation errors. This integration helps catch issues early in the development lifecycle.

**4. Code Analysis and Quality Assurance**
Static code analysis tools often leverage the Java Compiler to perform in-depth analyses of the codebase. These tools can identify potential issues, enforce coding standards, and provide insights into code complexity. The compiler-generated bytecode serves as a foundation for various code quality metrics and analyses.

## Significance of the Java Compiler

**1. Cross-Platform Compatibility**
The Java Compiler plays a crucial role in achieving Java's platform independence. By translating source code into bytecode, the compiler enables Java applications to run on any device or platform with a compatible Java Virtual Machine (JVM). This cross-platform compatibility is a key factor in Java's ubiquity across diverse computing environments.

**2. Productivity and Development Speed**
The Java Compiler contributes to the productivity of developers by providing rapid feedback during the coding phase. As developers write code, the compiler highlights syntax errors and issues, allowing for immediate correction. Additionally, the compiler's ability to generate bytecode quickly supports agile development practices, where code changes can be validated rapidly.

**3. Optimization and Performance**
Through various optimization techniques, the Java Compiler enhances the performance of Java applications. The generated bytecode benefits from optimizations that improve execution speed and reduce resource consumption. The ongoing evolution of the compiler includes advancements aimed at further optimizing Java applications in terms of both startup time and runtime performance.

**4. Enforcement of Language Constraints**
The Java Compiler enforces language constraints defined by the Java language specification. It ensures that the code adheres to syntactic and semantic rules, reducing the likelihood of runtime errors. By catching errors early in the development process, the compiler contributes to the creation of robust and reliable Java applications.

**5. Facilitating Code Analysis and Tools**
The Java Compiler generates bytecode that serves as the foundation for various code analysis tools, static analyzers, and IDE features. These tools leverage the compiler's output to provide insights into code quality, identify potential issues, and offer automated refactorings. The compiler's role extends beyond code generation to supporting a rich ecosystem of development tools.

## Challenges and Future Directions

**1. Compilation Speed and Efficiency**
As software projects grow in size and complexity, compilation speed becomes a critical factor in developer productivity. Efforts are ongoing to improve the speed and efficiency of the Java Compiler, including advancements in incremental compilation and the exploration of parallel compilation strategies.

**2. Integration with Modern Development Practices**
The Java Compiler continues to adapt to modern development practices, including the rise of microservices, cloud-native architectures, and containerization. Integration with build tools, container orchestration platforms, and cloud deployment scenarios remains an area of focus to streamline the development and deployment processes.

**3. Enhancements in Ahead-of-Time Compilation**
The exploration of Ahead-of-Time (AOT) compilation, as introduced with GraalVM, represents a potential shift in the way Java applications are compiled and executed. AOT compilation has the potential to improve startup times and reduce the runtime overhead associated with the Java Virtual Machine.

**4. Language Features and Compatibility**
As the Java language evolves with each new version, the Java Compiler must stay in sync to support new language features and enhancements. This includes support for features introduced in recent versions, such as records, pattern matching, and sealed classes. Compiler updates are essential to unlock the full potential of the Java language for developers.

## Conclusion

The Java Compiler stands as a fundamental and integral part of the Java development toolkit. It plays a multifaceted role in the software development lifecycle, from the initial stages of code creation to the final steps of deployment. The continuous evolution of the Java Compiler reflects the commitment to enhancing the performance, efficiency, and capabilities of Java applications. As developers navigate the intricacies of Java development, the compiler remains a reliable companion, transforming high-level Java code into bytecode and enabling the realization of Java's core principles of platform independence and "Write Once, Run Anywhere."

# M.Sc. (Computer Science)
# SEMESTER-3
# COURSE: Web Programming

| UNIT 2: FUNDAMENTALS OF JAVA |
|---|
| 2.1 Java Vs. C++ |
| 2.2 Byte Code |
| 2.3 Java Virtual Machine |
| 2.4 Constants |
| 2.5 Variables |
| 2.6 Data types |
| 2.7 Operators |
| 2.8 Expressions |
| 2.9 Control structures |
| 2.10 Defining class and Creating objects |
| 2.11 Accessing class members |
| 2.12 Constructors |
| 2.13 Garbage Collection |
| 2.14 Method overloading |

## 2.1 Java Vs. C++

**Introduction**

Java and C++ are two influential and widely-used programming languages that have shaped the landscape of software development. Both languages have their strengths, weaknesses, and unique features, making them suitable for different types of applications and scenarios. In this exploration, we will delve into the fundamentals of Java and C++, comparing key aspects such as syntax, memory management, performance, platform independence, and community support.

### I. Syntax and Language Features

#### 1. Java Syntax

**a. Simplicity and Readability:**

Java is known for its clean and readable syntax. It was designed with a focus on simplicity and ease of use, making it an ideal language for developers who value readability and maintainability. Java's syntax is influenced by C and C++, but it eliminates some of the more complex features of C++, aiming for a more straightforward and approachable language.

**b. Object-Oriented Paradigm:**

Java is a fully object-oriented programming language. Everything in Java is treated as an object, and the language supports encapsulation, inheritance, and polymorphism as core principles. This object-oriented nature contributes to modular and organized code structures.

**c. Memory Management:**

Java incorporates automatic memory management through a garbage collector. Developers don't need to explicitly manage memory allocation and deallocation, reducing the likelihood of memory leaks and segmentation faults. This simplifies the development process but can introduce some overhead.

### 2. C++ Syntax

**a. Versatility and Expressiveness:**
C++ is known for its versatility and expressiveness. It provides low-level features for systems programming while also supporting high-level abstractions through object-oriented programming (OOP) and generic programming. C++ syntax can be more complex than Java, offering a broad range of features for developers who require fine-grained control over memory and system resources.

**b. Multiple Paradigms:**
C++ supports multiple programming paradigms, including procedural, object-oriented, and generic programming. This versatility allows developers to choose the paradigm that best fits their needs, making C++ suitable for a wide range of applications.

**c. Memory Management:**
Unlike Java, C++ requires manual memory management. Developers have explicit control over memory allocation and deallocation, which can lead to more efficient memory usage but also introduces the risk of memory-related errors, such as memory leaks and segmentation faults.

## II. Memory Management

### 1. Java Memory Management

**a. Garbage Collection:**
Java employs automatic garbage collection to manage memory. The garbage collector identifies and deallocates objects that are no longer reachable, simplifying memory management for developers. While this approach reduces the risk of memory-related bugs, it can introduce some performance overhead.

**b. Memory Safety:**
Java's memory management model enhances memory safety by eliminating issues such as dangling pointers and memory leaks. Developers do not have direct access to memory addresses, reducing the likelihood of runtime errors related to manual memory management.

### 2. C++ Memory Management

**a. Manual Memory Management:**
C++ requires explicit memory management, and developers are responsible for allocating and deallocating memory using operators like `new` and `delete` or using smart pointers. While this manual control offers flexibility, it also introduces the potential for memory leaks and segmentation faults if not handled carefully.

**b. Efficiency and Control:**
C++'s manual memory management allows for fine-grained control over memory, leading to potentially more efficient resource usage. However, this efficiency comes at the cost of increased complexity and the need for careful attention to memory-related issues.

## III. Platform Independence

### 1. Java's Platform Independence

**a. Java Virtual Machine (JVM):**
Java achieves platform independence through the use of the Java Virtual Machine (JVM). Java source code is compiled into bytecode, which is then executed by the JVM. As long as a JVM is available for a specific platform, Java programs can run without modification, adhering to the "Write Once, Run Anywhere" (WORA) principle.

**b. Portability:**
Java applications are highly portable, allowing developers to write code on one platform and run it on various platforms without recompilation. This portability is a significant advantage in scenarios where

applications need to run on diverse environments.

### 2. C++ Platform Dependence

**a. Compilation to Native Code:**
C++ code is typically compiled to native machine code specific to the target platform. While this approach can result in optimized and efficient executables, it also means that C++ programs need to be recompiled for each target platform, making them less portable compared to Java.

**b. Platform-Specific Considerations:**
C++ developers often need to consider platform-specific details, such as different compilers, libraries, and system architectures. This can lead to additional effort in maintaining and ensuring compatibility across various platforms.

## IV. Performance Considerations

### 1. Java Performance Characteristics

**a. Just-In-Time Compilation:**
Java uses Just-In-Time (JIT) compilation to convert bytecode into native machine code at runtime. While this introduces a slight overhead during startup, it can result in optimized performance during the execution of long-running applications.

**b. Runtime Performance:**
Java's runtime performance has improved significantly over the years, but it may still be perceived as slightly slower than natively compiled languages like C++. However, the performance difference is often negligible for many applications, especially considering the benefits of platform independence.

### 2. C++ Performance Characteristics

**a. Ahead-of-Time Compilation:**
C++ programs are typically compiled ahead of time into native machine code. This can lead to faster startup times compared to Java, as there is no need for JIT compilation. Additionally, C++ allows for optimizations that can result in highly efficient executables.

**b. Fine-Grained Control:**
C++ provides developers with fine-grained control over low-level details, allowing for manual memory management and optimizations. This control can lead to highly performant code, especially in scenarios where maximum efficiency is crucial.

## V. Community Support and Ecosystem

### 1. Java Ecosystem

**a. Mature Libraries and Frameworks:**
Java boasts a mature and extensive ecosystem of libraries and frameworks. The Java Standard Edition (SE) and Java Enterprise Edition (EE) provide a rich set of APIs for various application domains. Additionally, popular frameworks like Spring, Hibernate, and Apache Camel contribute to the robustness of the Java ecosystem.

**b. Community Engagement:**
Java has a large and active community of developers, which facilitates knowledge sharing, collaboration, and the creation of open-source projects. The community-driven nature of Java has contributed to the language's enduring popularity and adaptability.

### 2. C++ Ecosystem

**a. Diverse Libraries and Tools:**
C++ also has a diverse ecosystem with a multitude of libraries and tools. The Standard Template

Library (STL) is a powerful collection of template classes and functions, and there are numerous third-party libraries for tasks ranging from graphics programming to numerical computation.

**b. Specialized Communities:**
C++ is used extensively in domains such as systems programming, game development, and high-performance computing. While the C++ community may be more specialized than Java's, it is highly engaged and contributes to the development of tools and libraries tailored to specific needs.

**Conclusion**

Java and C++ are both powerful programming languages with distinct characteristics that cater to different development needs. The choice between Java and C++ depends on various factors, including the nature of the project, development goals, and specific requirements. Java's platform independence, memory safety, and robust ecosystem make it an excellent choice for enterprise applications, web development, and mobile applications. On the other hand, C++'s fine-grained control over memory, high performance, and versatility make it a preferred language for systems programming, game development, and scenarios where maximum efficiency is crucial. Understanding the fundamental differences between Java and C++ empowers developers to make informed decisions based on the specific demands of their projects and the trade-offs involved in each language.

# 2.2 Byte Code

**Definition:**

Byte code is an intermediate code generated by a compiler after translating the source code written in a high-level programming language into a low-level, platform-independent representation. It is a set of instructions that is not directly executed by the CPU but is instead interpreted or executed by a virtual machine.

**Purpose:**

The use of byte code is a key feature in achieving platform independence in Java. Instead of generating native machine code for each platform, Java compilers generate byte code, which can be executed by any device or system that has a Java Virtual Machine (JVM).

**Characteristics:**

**1. Platform Independence:** Byte code is designed to be platform-independent, allowing Java programs to run on any device or operating system with a compatible JVM.

**2. Compact and Efficient:** Byte code is generally more compact than native machine code, making it easier to distribute and download Java programs over the internet. It is also designed for efficient interpretation by the JVM.

**3. Security:** Byte code is considered more secure than native code because it is executed by the JVM, which provides a layer of abstraction and security checks, preventing certain types of malicious activities.

# 2.3 Java Virtual Machine (JVM):

**Definition:**

The Java Virtual Machine (JVM) is a virtual machine that provides an execution environment for Java applications. It interprets and executes Java byte code, making it possible for Java programs to run on any device or platform that has a compatible JVM.

**Key Components:**

**1. Class Loader:** Responsible for loading classes into the JVM at runtime. Classes can be loaded

from the local file system, network, or other sources.

**2. Byte Code Verifier:** Ensures that the loaded byte code adheres to the Java Virtual Machine specification, preventing the execution of potentially harmful code.

**3. Interpreter:** Interprets and executes the Java byte code line by line. While interpretation is generally slower than direct execution of native code, it allows for platform independence.

**4. Just-In-Time (JIT) Compiler:** Translates frequently executed portions of byte code into native machine code for improved performance. This compilation happens at runtime.

**5. Garbage Collector:** Manages memory by reclaiming unused objects and freeing up resources, preventing memory leaks.

**6. Execution Engine:** Executes the byte code, either through interpretation or compilation to native code, depending on the JVM implementation.

**JVM Advantages:**

**1. Platform Independence:** JVM allows Java programs to be run on any device or operating system that has a compatible JVM installed.

**2. Memory Management:** JVM automates memory management, reducing the risk of memory leaks and making it easier for developers to write robust applications.

**3. Security:** The JVM provides a secure execution environment by enforcing access controls, verifying byte code, and offering features like the sandbox for applets.

**4. Performance Optimization:** The JIT compiler enhances performance by translating frequently executed code into native machine code.

# 2.4 Constants:

**Definition:**

Constants in programming are values or expressions that remain unchanged during the execution of a program. They are used to store fixed values such as numbers, characters, or strings that do not change their values during the program's execution.

**Types of Constants:**

**1. Numeric Constants:** These include integers, floating-point numbers, and hexadecimal numbers. For example:

```
int age = 25;
float pi = 3.14;
```

**2. Character Constants:** Represent single characters and are enclosed in single quotes. For example:

```
char grade = 'A';
```

**3. String Constants:** Represent sequences of characters and are enclosed in double quotes. For example:

```
String message = "Hello, World!";
```

**4. Boolean Constants:** Represent the values `true` or `false`. For example:

```
boolean isJavaFun = true;
```

### Use of Constants:

**1. Readability:** Constants improve code readability by providing meaningful names for fixed values, making it easier for developers to understand the purpose of the values used in the program.

**2. Maintenance:** If a constant value needs to be changed, it can be done in one place (the constant declaration), affecting all occurrences in the code. This simplifies maintenance and reduces the risk of errors.

**3. Avoiding Magic Numbers:** Constants are used to avoid the use of "magic numbers" (hard-coded numerical values without explanation), which can be confusing and error-prone.

## 2.5 Variables:

### Definition:

A variable is a named storage location in a program that holds a value, and the value may change during the execution of the program. Variables provide a way to manipulate and store data in computer programs.

### Key Concepts:

**1. Declaration:** Before using a variable, it must be declared, specifying its data type and name. For example:

```
int count;
```

**2. Initialization:** Variables can be given an initial value at the time of declaration or later in the program. For example:

```
int count = 0; // Initialization at declaration
```

**3. Assignment:** The value of a variable can be changed through assignment statements. For example:

```
count = 10; // Assigning a new value to the variable
```

**4. Data Types:** Variables have data types that determine the type of values they can store, such as integers, floating-point numbers, characters, etc.

### Types of Variables:

**1. Local Variables:** Declared within a method or block of code and are only accessible within that scope.

**2. Instance Variables (Fields):** Belong to an object and are declared within a class but outside any method. They are unique to each instance of the class.

**3. Class Variables (Static Variables):** Belong to the class rather than any specific instance. They are declared using the `static` keyword.

### Scope and Lifetime:

- **Scope:** Refers to the region of the program where a variable can be accessed.
- **Lifetime:** Refers to the duration for which a variable exists in memory during program execution.

### Variable Naming Conventions:

- Variable names should be meaningful and reflect the purpose of the variable.
- Use camelCase for variable names (e.g., `totalAmount`, `employeeSalary`).
- Avoid using single-letter variable names, except for loop counters.

# 2.6 Data Types:

**Definition:**

In programming, a data type is a classification of data that determines the type of operations that can be performed on the data and the way the data is stored in memory. Java is a strongly-typed language, meaning that each variable must be declared with a specific data type.

**Primitive Data Types:**

**1. Integer Types:**
- `byte`: 8-bit signed integer (-128 to 127).
- `short`: 16-bit signed integer (-32,768 to 32,767).
- `int`: 32-bit signed integer (-2^31 to 2^31-1).
- `long`: 64-bit signed integer (-2^63 to 2^63-1).

**2. Floating-Point Types:**
- `float`: 32-bit floating-point (single precision

).
- `double`: 64-bit floating-point (double precision).

**3. Character Type:**
- `char`: 16-bit Unicode character.

**4. Boolean Type:**
- `boolean`: Represents true or false.

**Reference Data Types:**

**1. Class Types:** Objects created from classes.
*String str = new String("Hello");*

**2. Interface Types:** Similar to class types, representing a set of abstract methods.
*Runnable runnableObj = new MyRunnable();*

**3. Array Types:** Collections of elements of the same data type.
*int[] numbers = {1, 2, 3, 4, 5};*

**User-Defined Data Types:**

**1. Classes:** Developers can define their own classes, creating custom data types with properties and methods.

**2. Enumerations (Enums):** A special data type used to define collections of constants.
*enum Day {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY};*

**Type Casting:**

**- Implicit Casting (Widening):** Automatic conversion of a smaller data type to a larger data type.
*int intValue = 10;*
*double doubleValue = intValue; // Implicit casting from int to double*

**- Explicit Casting (Narrowing**): Manual conversion of a larger data type to a smaller data type, requiring explicit casting.
*double doubleValue = 10.5;*
*int intValue = (int) doubleValue; // Explicit casting from double to int*

# 2.7 Operators

### 1. Arithmetic Operators:

### Explanation:

Arithmetic operators perform mathematical operations on numeric values. In Java, the basic arithmetic operators include addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and modulus (`%`).

- Addition (`+`): Adds two operands.
  *int sum = operand1 + operand2;*

- Subtraction (`-`): Subtracts the right operand from the left operand.
  *int difference = operand1 - operand2;*

- Multiplication (`*`): Multiplies two operands.
  *int product = operand1 * operand2;*

- Division (`/`): Divides the left operand by the right operand.
  *int quotient = operand1 / operand2;*

- Modulus (`%`): Returns the remainder of the division of the left operand by the right operand.
  *int remainder = operand1 % operand2;*

### Numerical Examples:

Let's consider numerical examples to illustrate arithmetic operators:

```
int a = 10;
int b = 3;

// Addition
int sum = a + b; // Result: 13

// Subtraction
int difference = a - b; // Result: 7

// Multiplication
int product = a * b; // Result: 30

// Division
int quotient = a / b; // Result: 3

// Modulus
int remainder = a % b; // Result: 1
```

In this example, `a` and `b` are operands, and the respective operations demonstrate the use of arithmetic operators.

### 2. Relational Operators:

### Explanation:

Relational operators are used to compare values and return a boolean result. In Java, these operators include equal to (`==`), not equal to (`!=`), greater than (`>`), less than (`<`), greater than or equal to (`>=`), and less than or equal to (`<=`).

- Equal to (`==`): Returns `true` if the left operand is equal to the right operand.
  *boolean isEqual = operand1 == operand2;*

- Not equal to (`!=`): Returns `true` if the left operand is not equal to the right operand.
  *boolean isNotEqual = operand1 != operand2;*

- Greater than (`>`): Returns `true` if the left operand is greater than the right operand.
  *boolean isGreaterThan = operand1 > operand2;*

- Less than (`<`): Returns `true` if the left operand is less than the right operand.
  *boolean isLessThan = operand1 < operand2;*

- Greater than or equal to (`>=`): Returns `true` if the left operand is greater than or equal to the right operand.
  *boolean isGreaterOrEqual = operand1 >= operand2;*

- Less than or equal to (`<=`): Returns `true` if the left operand is less than or equal to the right operand.
  *boolean isLessOrEqual = operand1 <= operand2;*

**Numerical Examples:**

Consider the following numerical examples to illustrate relational operators:

```
int x = 5;
int y = 8;

// Equal to
boolean isEqual = (x == y); // Result: false

// Not equal to
boolean isNotEqual = (x != y); // Result: true

// Greater than
boolean isGreaterThan = (x > y); // Result: false

// Less than
boolean isLessThan = (x < y); // Result: true

// Greater than or equal to
boolean isGreaterOrEqual = (x >= y); // Result: false

// Less than or equal to
boolean isLessOrEqual = (x <= y); // Result: true
```

In these examples, the relational operators compare the values of `x` and `y`, producing boolean results.

**3. Logical Operators:**
**Explanation:**

Logical operators perform logical operations on boolean values. In Java, the logical operators include logical AND (`&&`), logical OR (`||`), and logical NOT (`!`).

- Logical AND (`&&`): Returns `true` if both the left and right operands are `true`.
  *boolean result = (operand1 && operand2);*

- Logical OR (`||`): Returns `true` if at least one of the operands is `true`.
  *boolean result = (operand1 || operand2);*

- Logical NOT (`!`): Returns `true` if the operand is `false`, and vice versa.
  *boolean result = !operand;*

**Numerical Examples:**

Let's use numerical examples to illustrate logical operators:

```
boolean a = true;
boolean b = false;

// Logical AND
boolean logicalAnd = (a && b); // Result: false

// Logical OR
boolean logicalOr = (a || b); // Result: true

// Logical NOT
boolean logicalNotA = !a; // Result: false
boolean logicalNotB = !b; // Result: true
```

In these examples, logical operators combine boolean values to produce logical results.

**4. Assignment Operators:**
**Explanation:**

Assignment operators are used to assign values to variables. In Java, the basic assignment operator is `=`. Additionally, compound assignment operators, such as `+=`, `-=`, `*=`, and `/=`, combine an operation with assignment.

- Assignment (`=`): Assigns the value of the right operand to the left operand.
  *variable = expression;*

- Compound Assignment (`+=`, `-=`, `*=`, `/=`): Combines an arithmetic operation with assignment.
  *variable += expression; // Equivalent to variable = variable + expression;*

**Numerical Examples:**

Consider the following numerical examples to illustrate assignment operators:

```
int c = 7;

// Assignment
c = 10; // c is now 10

// Compound Assignment
c += 3; // Equivalent to c = c + 3; // Result: 13

c -= 2; // Equivalent to c = c - 2; // Result: 11
```

*c *= 5; // Equivalent to c = c * 5; // Result: 55*

*c /= 2; // Equivalent to c = c / 2; // Result: 27*

In these examples, assignment and compound assignment operators are used to modify the value of the variable `c`.

**5. Increment and Decrement Operators:**
**Explanation:**

Increment and decrement operators are used to increase or decrease the value of a variable by 1. In Java, these operators are `++` (increment) and `--` (decrement). They can be used in both pre-increment/decrement and post-increment/decrement forms.

- Increment (`++`): Increases the value of the operand by 1.
  *variable++; // Post-increment*
  *++variable; // Pre-increment*

- Decrement (`--`): Decreases the value of the operand by 1.
  *variable--; // Post-decrement*
  *--variable; // Pre-decrement*

**Numerical Examples:**

Let's use numerical examples to illustrate increment and decrement operators:

*int counter = 0;*

*// Increment*
*counter++; // Post-increment // Result: 1*
*++counter; // Pre-increment // Result: 2*

*// Decrement*
*counter--; // Post-decrement // Result: 1*
*--counter; // Pre-decrement // Result: 0*

In these examples, increment and decrement operators modify the value of the variable `counter`.

**6. Bitwise Operators:**
**Explanation:**

Bitwise operators perform operations at the bit level, manipulating individual bits of integer values. In Java, the bitwise operators include bitwise AND (`&`), bitwise OR (`|`), bitwise XOR (`^`), and bitwise complement (`~`).

- Bitwise AND (`&`): Performs a bitwise AND operation between corresponding bits of the operands.
  *int result = operand1 & operand2;*

- Bitwise OR (`|`): Performs a bitwise OR operation between corresponding bits of the operands.
  *int result = operand1 | operand2;*

- Bitwise XOR (`^`): Performs a bitwise exclusive OR operation between corresponding bits of the operands.
  *int result = operand1 ^ operand2;*

- Bitwise Complement (`~`): Flips the bits of the operand, changing 1s to 0s and vice versa.
  *int result = ~operand;*

**Numerical Examples:**

Consider the following numerical examples to illustrate bitwise operators:

```
int x = 5; // Binary: 0101
int y = 3; // Binary: 0011

// Bitwise AND
int bitwiseAnd = x & y; // Result: 1 (Binary: 0001)

// Bitwise OR
int bitwiseOr = x | y; // Result: 7 (Binary: 0111)

// Bitwise XOR
int bitwiseXor = x ^ y; // Result: 6 (Binary: 0110)

// Bitwise Complement
int bitwiseComplementX = ~x; // Result: -6 (Binary: 11111111111111111111111111111010)
```

In these examples, bitwise operators manipulate the binary representations of integer values.

**7. Conditional (Ternary) Operator:**
**Explanation:**

The conditional operator, also known as the ternary operator, provides a concise way to write an if-else statement in a single line. It has the form `(condition) ? expression1 : expression2`, where `condition` is evaluated, and if it is `true`, `expression1` is executed; otherwise, `expression2` is executed.

*variable = (condition) ? expression1 : expression2;*

**Numerical Example:**

Consider the following numerical example to illustrate the conditional operator:

```
int a = 10;
int b = 5;

// Conditional Operator
int max = (a > b) ? a : b; // If a > b, assign a to max; otherwise, assign b.
```

In this example, the value of `max` will be assigned the larger of the two values, `a` or `b`, based on the condition `(a > b)`.

**Conclusion:**

Operators are fundamental elements in programming languages, allowing developers to perform various operations on data. Understanding the different categories of operators, their theoretical foundations, and numerical examples is crucial for writing effective and expressive code. Whether working with arithmetic, relational, logical, assignment, increment/decrement, bitwise, or conditional operators, developers can leverage these tools to build robust and efficient programs.

Certainly! Let's delve into each of the mentioned topics in Java, including expressions, control

structures, defining classes, creating objects, accessing class members, constructors, Garbage Collection, and method overloading.

# 2.8. Expressions in Java:

In Java, an expression is a combination of variables, operators, and method calls that produces a value. Expressions can be simple or complex and are fundamental to performing computations in a program.

**Explanation:**

**Arithmetic Expressions:**

- Arithmetic expressions involve mathematical operations using operators like `+`, `-`, `*`, `/`, and `%`.

*int result = 5 + 3 * 2; // Result: 11*

**Boolean Expressions:**

- Boolean expressions evaluate to either `true` or `false` based on conditions.

*boolean isGreater = (5 > 3); // Result: true*

**String Concatenation:**

- Combining strings using the `+` operator.

*String greeting = "Hello, " + "World!"; // Result: "Hello, World!"*

# 2.9 Control Structures in Java:

Control structures in Java govern the flow of execution in a program. Common control structures include conditional statements (`if`, `else`, `switch`) and looping statements (`for`, `while`, `do-while`).

**Explanation:**

**- if-else Statement:**

- Used for decision-making based on a condition.

```
int x = 10;
if (x > 5) {
    // Code to execute if condition is true
} else {
    // Code to execute if condition is false
}
```

**- switch Statement:**

- Used for multi-branching based on the value of an expression.

```
int day = 2;
switch (day) {
    case 1:
        // Code for Monday
        break;
    case 2:
        // Code for Tuesday
        break;
    // ... other cases
    default:
        // Code for other days
}
```

**- for Loop:**
- Executes a block of code a specified number of times.

```
for (int i = 0; i < 5; i++) {
    // Code to execute in each iteration
}
```

## 2.10. Defining Class and Creating Objects in Java:

In Java, a class is a blueprint for creating objects. Objects are instances of classes and encapsulate data and behavior.

**Explanation:**

**- Defining a Class:**
- A class definition includes fields (attributes) and methods (functions).

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

    void start() {
        // Code to start the car
    }
}
```

**- Creating Objects:**
- Objects are instances of a class and are created using the `new` keyword.

```
Car myCar = new Car();
myCar.model = "Toyota";
myCar.year = 2022;
```

## 2.11 Accessing Class Members in Java:

Accessing class members involves referring to the fields and methods of a class from within the class or outside it.

**Explanation:**

**- Accessing Fields:**
- Fields can be accessed using object references.

```
String carModel = myCar.model; // Accessing the model field
```

**- Accessing Methods:**
- Methods are called using object references.

```
myCar.start(); // Calling the start method
```

## 2.12 Constructors in Java:

Constructors are special methods in Java used for initializing objects. They have the same name as the class and are called when an object is created.

**Explanation:**

**- Default Constructor:**
- A constructor with no parameters.

```
public class Car {
    // Default constructor
    public Car() {
        // Initialization code
    }
}
```

**- Parameterized Constructor:**
- A constructor with parameters for initializing fields.

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

    // Parameterized constructor
    public Car(String carModel, int carYear) {
        model = carModel;
        year = carYear;
    }
}
```

## 2.13 Garbage Collection in Java:

Garbage Collection is a feature in Java that automatically reclaims memory occupied by objects that are no longer in use.

**Explanation:**

**- Automatic Memory Management:**
- Java's Garbage Collector automatically identifies and deletes unreferenced objects.

```
Car myCar = new Car();
// ... Code that no longer references myCar
// Garbage Collector reclaims memory used by myCar
```

## 2.14 Method Overloading in Java:

Method overloading allows a class to have multiple methods with the same name but different parameters.

**Explanation:**

**- Overloaded Methods:**
- Methods with the same name but different parameter lists.

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

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

**- Calling Overloaded Methods:**

*- The correct method is called based on the arguments provided.*

```
Calculator calc = new Calculator();
int sumInt = calc.add(5, 3); // Calls the int version
double sumDouble = calc.add(5.5, 3.7); // Calls the double version
```

## Conclusion:

In this comprehensive overview, we covered key concepts in Java programming, including expressions, control structures, defining classes, creating objects, accessing class members, constructors, Garbage Collection, and method overloading. Understanding these concepts is essential for building robust and effective Java applications. Numerical examples and coding illustrate the practical application of these concepts, fostering a deeper comprehension of Java programming fundamentals.

# M.Sc. (Computer Science)
# SEMESTER-3
# COURSE: Web Programming

| UNIT 3: INHERITANCE |
|---|
| 3.1 Different types of Inheritance |
| 3.2 Member access |
| 3.3 Using super keyword to call super class constructors |
| 3.4 Creating a multilevel hierarchy |
| 3.5 Method overriding |
| 3.6 Dynamic method dispatch |
| 3.7 Using abstract classes |
| 3.8 Using Final keyword |

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (subclass or derived class) to inherit properties and behaviors from another class (superclass or base class). This facilitates code reuse, extensibility, and the creation of a hierarchical class structure. In Java, inheritance is implemented through the `extends` keyword.

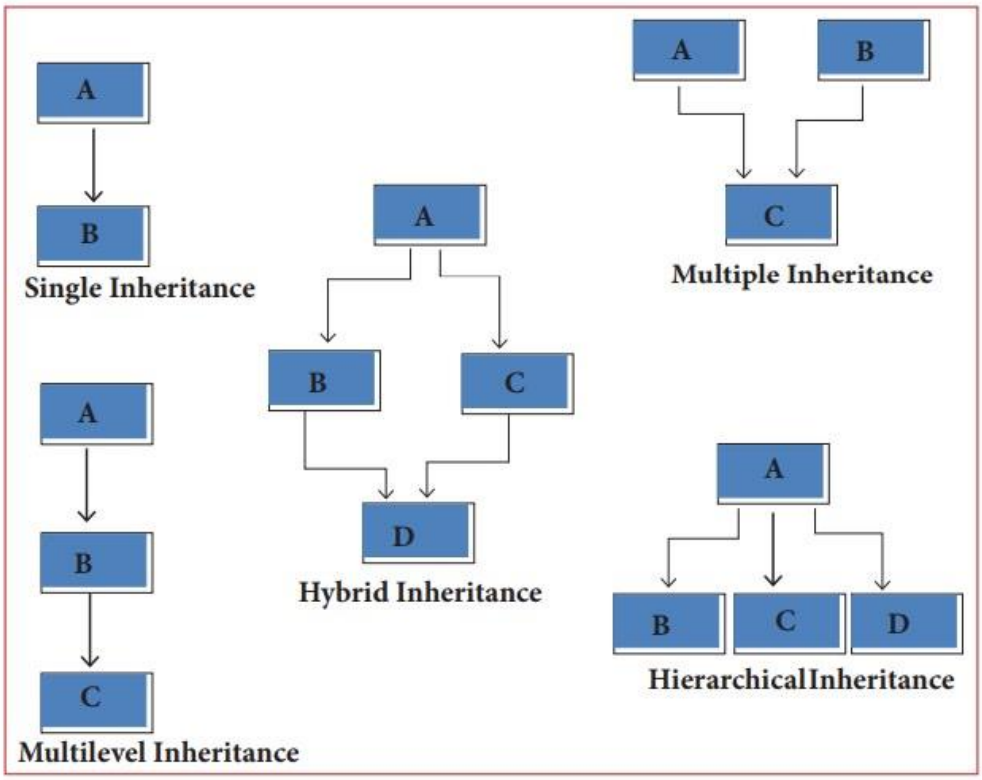
## 3.1 Types of Inheritance:

**1. Single Inheritance:**

- In single inheritance, a class extends only one superclass. The subclass inherits the fields and methods of the single superclass.

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

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




**2. Multiple Inheritance (via Interfaces):**

- Java supports multiple inheritance through interfaces. A class can implement multiple interfaces, allowing it to inherit from multiple sources.

```
interface Swim {
    void swim();
}

class Fish implements Animal, Swim {
    // Fish can inherit from both Animal and Swim
}
```

Note: Direct multiple inheritance with classes is not supported in Java to avoid the "diamond problem."

**3. Multilevel Inheritance:**

- In multilevel inheritance, a class derives from another class, and then another class derives from it. This creates a chain of inheritance.

```
class Grandparent {
    // Grandparent class members
}

class Parent extends Grandparent {
    // Parent class members
}

class Child extends Parent {
    // Child class members
}
```

**4. Hierarchical Inheritance:**

- In hierarchical inheritance, multiple classes derive from a single superclass. Each subclass inherits from the same superclass.

```
class Vehicle {
    // Vehicle class members
}

class Car extends Vehicle {
    // Car class members
}

class Motorcycle extends Vehicle {
    // Motorcycle class members
}
```

# 3.2 Member Access in Inheritance:

**- Public Members:**

- Public members of the superclass are accessible in the subclass.

```
class Animal {
    public void eat() {
        System.out.println("Animal is eating");
    }
}

class Dog extends Animal {
    void dogAction() {
        eat(); // Accessing public method from superclass
    }
}
```

**- Protected Members:**

- Protected members are accessible within the package and by subclasses, promoting a level of encapsulation.

```
class Animal {
    protected void sleep() {
        System.out.println("Animal is sleeping");
    }
}

class Dog extends Animal {
    void dogAction() {
        sleep(); // Accessing protected method from superclass
    }
}
```

**- Default (Package-Private) Members:**

- Members with default access are accessible within the same package but not in subclasses outside the package.

```
class Animal {
    void move() {
        System.out.println("Animal is moving");
    }
}
```

```
class Dog extends Animal {
    void dogAction() {
        move(); // Accessing default method from superclass
    }
}
```

**- Private Members:**

- Private members are not directly accessible in subclasses. They are encapsulated within the superclass.

```
class Animal {
    private void breathe() {
        System.out.println("Animal is breathing");
    }
}

class Dog extends Animal {
    // Private method breathe() is not accessible in Dog class
}
```

**Numerical Examples and Coding:**

Let's explore numerical examples and code snippets to illustrate inheritance in Java:

**1. Single Inheritance:**

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

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

public class InheritanceExample {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.eat(); // Inherited method
        myDog.bark(); // Subclass method
    }
}
```

**2. Multiple Inheritance (via Interfaces):**

```
interface Swim {
    void swim();
}

class Fish implements Animal, Swim {
    @Override
    public void eat() {
        System.out.println("Fish is eating");
```

```
    }

    @Override
    public void swim() {
        System.out.println("Fish is swimming");
    }
}
```

**3. Multilevel Inheritance:**

```
class Grandparent {
    void grandparentAction() {
        System.out.println("Grandparent action");
    }
}

class Parent extends Grandparent {
    void parentAction() {
        System.out.println("Parent action");
    }
}

class Child extends Parent {
    void childAction() {
        System.out.println("Child action");
    }
}
```

**4. Hierarchical Inheritance:**

```
class Vehicle {
    void move() {
        System.out.println("Vehicle is moving");
    }
}

class Car extends Vehicle {
    void carAction() {
        System.out.println("Car action");
    }
}

class Motorcycle extends Vehicle {
    void motorcycleAction() {
        System.out.println("Motorcycle action");
    }
}
```

**Conclusion:**

Inheritance in Java is a powerful mechanism that promotes code reuse and the creation of modular, extensible code. Different types of inheritance, including single, multiple (via interfaces), multilevel, and hierarchical, provide flexibility in designing class hierarchies. Understanding member access, including public, protected, default, and private members, is crucial for proper encapsulation and managing the visibility of class members. By leveraging inheritance, developers can build scalable and maintainable Java applications with a hierarchical structure that reflects the relationships between various entities in the system.

## 3.3 Using `super` Keyword to Call Superclass Constructors:

In Java, the `super` keyword is used to refer to the superclass (parent class) members, including fields, methods, and constructors. When dealing with constructors, `super()` is used to invoke the constructor of the superclass. This is particularly useful in scenarios where the subclass wants to initialize its own attributes along with those inherited from the superclass.

**Explanation:**

```
class Animal {
    int age;

    Animal(int age) {
        this.age = age;
    }
}

class Dog extends Animal {
    String breed;

    Dog(int age, String breed) {
        super(age); // Calling the constructor of the superclass
        this.breed = breed;
    }
}
```

In this example, the `Dog` class extends the `Animal` class. The `Dog` class has its own field `breed`, and the `super(age)` call ensures that the `age` attribute from the `Animal` class is also initialized.

## 3.4 Creating a Multilevel Hierarchy:

Multilevel inheritance in Java involves a chain of inheritance where a subclass extends another class, and another subclass extends the previous subclass. This creates a hierarchical structure with multiple levels.

**Explanation:**

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

class Mammal extends Animal {
    void walk() {
        System.out.println("Mammal is walking");
    }
}

class Dog extends Mammal {
    void bark() {
        System.out.println("Dog is barking");
    }
```

```
}
```

In this example, `Animal` is the base class, `Mammal` inherits from `Animal`, and `Dog` inherits from `Mammal`. This creates a multilevel hierarchy where `Dog` has access to both `eat()` and `walk()` methods.

## 3.5 Method Overriding:

Method overriding is a feature in Java that allows a subclass to provide a specific implementation for a method that is already defined in its superclass. This enables polymorphism, where a reference to the superclass can be used to refer to objects of the subclass.

**Explanation:**

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

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

Here, the `Dog` class overrides the `sound()` method defined in the `Animal` class. When an object of type `Dog` calls the `sound()` method, the overridden version in the `Dog` class is executed.

## 3.6 Dynamic Method Dispatch:

Dynamic method dispatch is a mechanism where the correct method to be called is determined at runtime based on the type of the object. It is a crucial aspect of polymorphism in Java.

**Explanation:**

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

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

public class TestPolymorphism {
    public static void main(String[] args) {
        Animal myAnimal = new Dog(); // Upcasting
        myAnimal.sound(); // Calls the overridden sound() in Dog
```

```
    }
}
```

In this example, an object of type `Dog` is assigned to a variable of type `Animal`. During runtime, the `sound()` method of the `Dog` class is called, demonstrating dynamic method dispatch.

## 3.7 Using Abstract Classes:

An abstract class in Java is a class that cannot be instantiated and may contain abstract methods, which are methods without a body. Subclasses must provide implementations for these abstract methods.

**Explanation:**

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

    void display() {
        System.out.println("Displaying shape");
    }
}

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

In this example, `Shape` is an abstract class with an abstract method `draw()`. The `Circle` class extends `Shape` and provides an implementation for the `draw()` method.

## 3.8 Using `final` Keyword:

The `final` keyword in Java is used to restrict the modification of classes, methods, and variables. A `final` class cannot be inherited, a `final` method cannot be overridden, and a `final` variable cannot be reassigned.

**Explanation:**

```
final class ImmutableClass {
    // Class definition
}

class MyClass {
    final int constantValue = 10; // Final variable

    final void display() {
        // Method with final keyword
    }
}
```

In this example, `ImmutableClass` is declared as `final`, preventing any further inheritance. The `constantValue` variable and the `display()` method are marked as `final`, indicating that their values

or implementations cannot be changed.

## Conclusion:

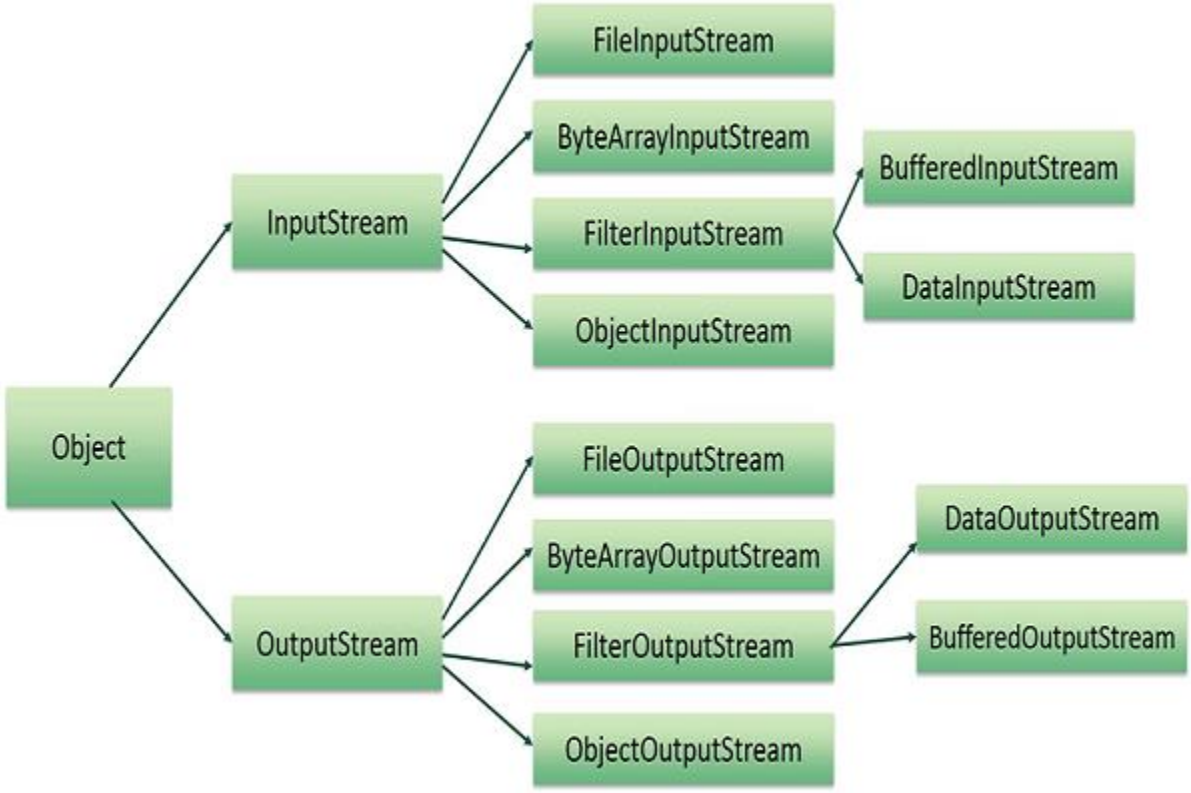
In this comprehensive explanation, we covered essential Java concepts, including the use of the `super` keyword to call superclass constructors, creating a multilevel hierarchy, method overriding, dynamic method dispatch, using abstract classes, and employing the `final` keyword. Understanding these concepts is fundamental for developing robust and flexible Java applications. The provided theoretical explanations, numerical examples, and code snippets offer a practical understanding of how these concepts are implemented and utilized in real-world scenarios. Developers can leverage these concepts to enhance code organization, promote code reuse, and build more maintainable and scalable software.

# M.Sc. (Computer Science)
# SEMESTER-3
# COURSE: Web Programming

| UNIT 4: I/O BASICS |
|---|
| 4.1 Streams and the predefined streams |
| 4.2 Reading console Input |
| 4.3 Writing console Output |
| 4.4 Arrays and Strings: One-dimensional and two-dimensional Arrays |
| 4.5 String Handling using String, String Buffer class, and String Functions |

## 4.1 Streams in Java:

Streams in Java are abstractions that provide a consistent way to perform input and output operations. They are classified into two types: input streams for reading data and output streams for writing data.




**a. Input Streams (`System.in`):**

`System.in` represents the standard input stream, typically connected to the keyboard. Reading from the console can be done using classes like `Scanner` or `BufferedReader`.

**Example:**

```
import java.util.Scanner;

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

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

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

        scanner.close();
    }
}
```

In this example, `Scanner` is used to read input from the console. The `nextLine()` method reads the entire line entered by the user.

**b. Output Streams (`System.out`):**
`System.out` represents the standard output stream, usually connected to the console. Writing to the console is done using methods like `println()`.

**Example:**

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

        System.out.println(message);
    }
}
```

The `println()` method prints the specified message to the console, followed by a newline character.

## 4.2 Reading Console Input in Java:

To read console input in Java, you can use the `Scanner` class. The `Scanner` class is part of the `java.util` package, and it provides methods to read various types of data from the console.

Here is a simple example of reading a string from the console:

```
import java.util.Scanner;

public class ConsoleInputExample {
    public static void main(String[] args) {
        // Create a Scanner object to read input from the console
        Scanner scanner = new Scanner(System.in);

        // Prompt the user to enter a string
        System.out.print("Enter a string: ");

        // Read the entered string
        String userInput = scanner.nextLine();

        // Display the entered string
        System.out.println("You entered: " + userInput);

        // Close the Scanner to avoid resource leaks
        scanner.close();
    }
```

```
}
```

## 4.3 Writing Console Output in Java:

To write console output in Java, you can use the `System.out.println` or `System.out.print` methods. These methods are part of the `System` class and are used to print text to the console.

Here is a simple example of writing output to the console:

```
public class ConsoleOutputExample {
    public static void main(String[] args) {
        // Display a message using println
        System.out.println("This is a simple console output example.");

        // Display multiple messages without moving to the next line using print
        System.out.print("Hello, ");
        System.out.print("world!");

        // Print an empty line
        System.out.println();

        // Display numeric values
        int number = 42;
        double pi = 3.14;
        System.out.println("Number: " + number + ", Pi: " + pi);
    }
}
```

## 4.4 Arrays and Strings:

**1. One-dimensional Arrays:**
A one-dimensional array in Java is a collection of elements of the same type arranged in a linear order.

**a. Declaration and Initialization:**

```
int[] numbers = new int[5];
numbers[0] = 1;
numbers[1] = 2;
// ...

// Shortened Syntax
int[] numbers = {1, 2, 3, 4, 5};
```

In Java, arrays are zero-indexed, meaning the first element is at index 0.

**b. Accessing Elements:**

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

The `length` property of an array provides the number of elements it contains.

**2. Two-dimensional Arrays:**
A two-dimensional array is an array of arrays, creating a grid-like structure.

**a. Declaration and Initialization:**
```
int[][] matrix = new int[3][3];
matrix[0][0] = 1;
matrix[0][1] = 2;
// ...

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

**b. Accessing Elements:**
```
for (int i = 0; i < matrix.length; i++) {
   for (int j = 0; j < matrix[i].length; j++) {
      System.out.print(matrix[i][j] + " ");
   }
   System.out.println();
}
```

Nested loops are used to iterate through each element in a two-dimensional array.

# 4.5 String Handling:

**a. Using `String` Class:**
The `String` class in Java represents a sequence of characters and is immutable.

**Example:**
```
String text = "Hello, Java!";

// Length of the string
int length = text.length(); // Returns 13

// Concatenation
String greeting = text.concat(" Welcome!"); // "Hello, Java! Welcome!"

// Substring
String subString = text.substring(0, 5); // "Hello"

// Comparison
boolean isEqual = text.equals("Hello, Java!"); // true

// Searching
int index = text.indexOf("Java"); // Returns the index of "Java" in the string
```

**b. Using `StringBuffer` Class:**
The `StringBuffer` class in Java is mutable, allowing modifications to the content.

**Example:**
```
StringBuffer buffer = new StringBuffer("Hello");

// Append
buffer.append(", Java!"); // "Hello, Java!"
```

*// Insert*
*buffer.insert(5, " World"); // "Hello World, Java!"*

*// Delete*
*buffer.delete(0, 5); // Deletes "Hello"*

*// Reverse*
*buffer.reverse(); // Reverses the string*
```

*`StringBuffer` provides methods for modifying the content of the string.*

**Conclusion:**

In this detailed exploration, we covered the foundational concepts of I/O operations, arrays, and strings in Java. Understanding I/O streams, array manipulation, and string handling is essential for developing versatile and interactive Java applications. The provided examples and code snippets aim to reinforce these concepts, providing a practical understanding for developers. Leveraging these fundamental skills will enable developers to build efficient, readable, and maintainable Java code.
```

# M.Sc. (Computer Science)
# SEMESTER-3
# COURSE: Web Programming

| UNIT 5: PACKAGES |
|---|
| 5.1 Types of packages |
| 5.2 Defining a package |
| 5.3 Importing packages |
| 5.4 Access protection Interfaces: Defining an Interface |
| 5.5 Implementing Interfaces and Variables in Interfaces |
| 5.6 Achieving multiple inheritance using interfaces |
| 5.7 Interface and Abstract classes |

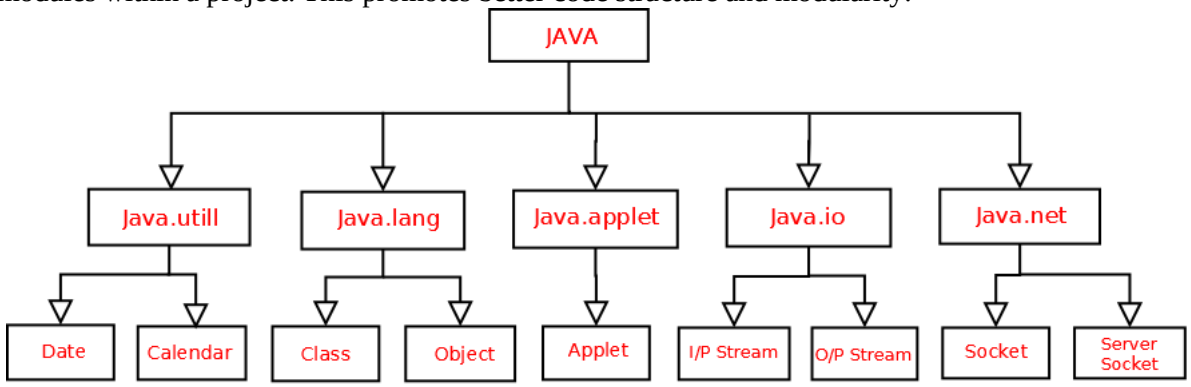
**Introduction to Packages:**

In Java, packages are used to organize classes and interfaces into a hierarchical structure, providing a way to encapsulate code and prevent naming conflicts. They contribute to code organization, reusability, and maintenance.

## 5.1 Types of Packages:

**a. Built-in Packages:**
Java provides a set of built-in packages, such as `java.lang`, `java.util`, and `java.io`. These packages contain fundamental classes and utilities that are commonly used in Java programs.

**b. User-Defined Packages:**
Developers can create their own packages to organize classes related to specific functionalities or modules within a project. This promotes better code structure and modularity.




## 5.2 Defining a Package:

To define a package in Java, you include a `package` statement at the beginning of your Java source file. The package statement is followed by the package name.

**1. Defining a Package Example:**

Suppose we want to create a package named `com.example.util` for utility classes.

```
// File: UtilClass.java
package com.example.util;

public class UtilClass {
    // Class implementation
}
```

This `UtilClass` is now part of the `com.example.util` package.

## 5.3 Importing Packages:

In Java, the `import` statement is used to bring classes or entire packages into scope, allowing their usage without fully qualified names.

1. Importing a Class Example:

```
// File: MyClass.java
import com.example.util.UtilClass;

public class MyClass {
    public static void main(String[] args) {
        UtilClass util = new UtilClass();
        // Use util object
    }
}
```

Here, `UtilClass` from the `com.example.util` package is imported and used within the `MyClass` class.

2. Importing an Entire Package Example:

```
// File: AnotherClass.java
import com.example.util.*;

public class AnotherClass {
    public static void main(String[] args) {
        UtilClass util = new UtilClass();
        // Use util object
    }
}
```

The `*` wildcard imports all classes from the `com.example.util` package.

## 5.4 Access Protection:

Access protection in Java is managed through access modifiers (`public`, `protected`, default, and `private`). These modifiers determine the visibility of classes, methods, and variables.

**1. Access Modifiers:**

**a. Public:**
Public members are accessible from any class.

**b. Protected:**
Protected members are accessible within the same package and by subclasses (even if they are in a

different package).

**c. Default (Package-Private):**
Members with default access are accessible only within the same package.

**d. Private:**
Private members are accessible only within the same class.

**2. Access Protection Example:**

Consider a package `com.example.access` with two classes, `ClassA` and `ClassB`.

```
// File: ClassA.java
package com.example.access;

public class ClassA {
    public int publicVar;
    protected int protectedVar;
    int defaultVar;
    private int privateVar;
}

// File: ClassB.java
package com.example.access;

public class ClassB extends ClassA {
    public static void main(String[] args) {
        ClassA a = new ClassA();
        a.publicVar = 1;       // Accessible
        a.protectedVar = 2;    // Accessible within the same package or subclasses
        a.defaultVar = 3;      // Accessible within the same package
        // a.privateVar = 4;   // Not accessible

        ClassB b = new ClassB();
        b.protectedVar = 5;    // Accessible because ClassB is a subclass
    }
}
```

In this example, `ClassA` has members with different access modifiers. `ClassB` is a subclass of `ClassA` and can access protected members.

# 5.5 Introduction to Interfaces:

Interfaces in object-oriented programming provide a way to define a contract that classes must adhere to. They serve as blueprints for classes, specifying a set of methods that implementing classes must provide. Interfaces play a pivotal role in achieving abstraction, multiple inheritance, and ensuring a consistent structure in diverse class hierarchies.

**Defining an Interface:**

An interface is typically declared using a specific keyword in a programming language. For instance, in Java:

```
interface Shape {
   double calculateArea();
   void display();
}
```

Here, the `Shape` interface declares two methods: `calculateArea()` and `display()`. Any class that implements this interface must furnish concrete implementations for these methods.

**Implementing Interfaces:**

To implement an interface in a class, the `implements` keyword is used:

```
class Circle implements Shape {
   double radius;

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

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

   public void display() {
      System.out.println("Circle with radius " + radius);
   }
}
```

The `Circle` class, in this case, implements the `Shape` interface, providing concrete definitions for the methods specified in the interface.

**Variables in Interfaces:**

Interfaces can also include variables, which are implicitly `public`, `static`, and `final`. These variables act as constants shared among all classes implementing the interface:

```
interface Shape {
   double PI = 3.14159;
   double calculateArea();
   void display();
}
```

Now, any class implementing `Shape` can access the constant using `Shape.PI`.

## 5.6 Achieving Multiple Inheritance Using Interfaces:

Interfaces facilitate the achievement of multiple inheritance, allowing a class to implement multiple interfaces. This is particularly beneficial in scenarios where a class needs to inherit functionalities from different sources:

```
class ColoredCircle implements Shape, Color {
   // ... (Implementation details)
}
```

Here, `ColoredCircle` implements both the `Shape` and `Color` interfaces, inheriting behaviors from both.

## 5.7 Interface and Abstract Classes:

While interfaces provide a way to achieve abstraction and multiple inheritance, abstract classes also contribute to these goals. Abstract classes can contain a mix of abstract and concrete methods:

```
abstract class AbstractShape {
   double PI = 3.14159;

   void displayMessage() {
      System.out.println("This is a shape.");
   }

   abstract double calculateArea();
}
```

Classes can extend this abstract class, inheriting the constant and concrete method while providing implementations for the abstract method.

**Conclusion:**

Interfaces, with their ability to define contracts and support multiple inheritance, are a crucial tool in designing flexible and maintainable object-oriented systems. By using interfaces, developers can create modular and extensible code, promoting code reuse and ensuring a consistent structure across different classes. The combination of interfaces and abstract classes provides a versatile toolkit for building robust and scalable software architectures.

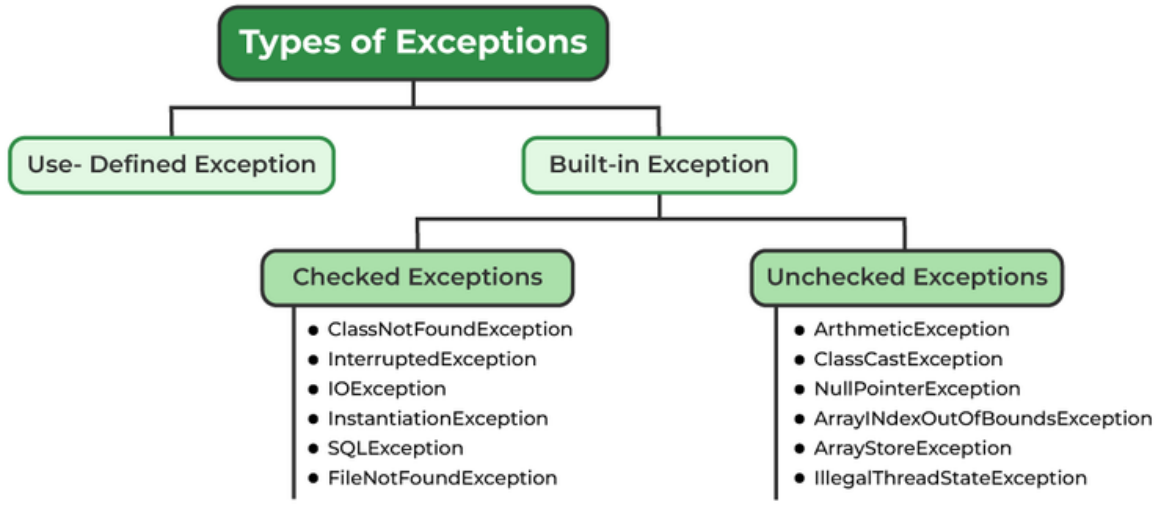
# M.Sc. (Computer Science)
# SEMESTER-3
# COURSE: Web Programming

| UNIT 6: EXCEPTION HANDLING |
|---|
| 6.1 Java Exception handling model |
| 6.2 Types of exception |
| 6.3 Using Try and catch |
| 6.4 Multiple Try and Catch clauses |
| 6.5 Nested Try statements |
| 6.6 Finally block |
| 6.7 User defined exceptions |

## 6.1 Exception Handling in Java:

Exception handling is a crucial aspect of Java programming that deals with runtime errors and exceptional conditions. It helps in writing robust and fault-tolerant code by providing a systematic way to handle unexpected situations.

**Java Exception Handling Model:**




In Java, exceptions are objects representing errors that can occur during program execution. The exception handling model involves the following key components:

- Try and Catch Blocks: The `try` block encloses a section of code where exceptions might occur. The `catch` block catches and handles specific exceptions that may arise within the `try` block.

- Multiple Try and Catch Clauses: A single `try` block can have multiple `catch` blocks to handle different types of exceptions independently.

- Nested Try Statements: `try` blocks can be nested within each other, allowing for more granular exception handling.

- Finally Block: The `finally` block contains code that will be executed whether an exception occurs or not. It is useful for releasing resources or performing cleanup operations.

- User-Defined Exceptions: Java allows developers to create their own exception classes, extending the standard exception classes to represent application-specific error conditions.

## 6.2 Types of Exceptions:

Java categorizes exceptions into two main types:

1. Checked Exceptions: These are exceptions that the compiler forces you to handle explicitly. They extend the `Exception` class. Examples include `IOException` and `ClassNotFoundException`.

2. Unchecked Exceptions: Also known as runtime exceptions, these exceptions extend the `RuntimeException` class. They are not checked by the compiler, and it is not mandatory to handle them. Examples include `NullPointerException` and `ArrayIndexOutOfBoundsException`.

## 6.3 Using Try and Catch:

Here's a simple example demonstrating the use of `try` and `catch` blocks to handle a potential division by zero exception:

```
public class ExceptionHandlingExample {
    public static void main(String[] args) {
        int numerator = 10;
        int denominator = 0;

        try {
            int result = numerator / denominator;
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.err.println("Error: " + e.getMessage());
        }
    }
}
```

In this example, if the denominator is zero, an `ArithmeticException` will be thrown, and the catch block will handle the exception, preventing the program from terminating abruptly.

## 6.4 Multiple Try and Catch Clauses:

```
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            String str = null;
            System.out.println(str.length());  // This will throw a NullPointerException
        } catch (NullPointerException e) {
            System.err.println("NullPointerException: " + e.getMessage());
        } catch (Exception e) {
            System.err.println("Generic Exception: " + e.getMessage());
        }
    }
}
```

In this example, the `NullPointerException` is caught by the first `catch` block. If a more generic exception handling is required, you can add a catch block with the `Exception` type.

## 6.5 Nested Try Statements:

```
public class NestedTryExample {
    public static void main(String[] args) {
        try {
            int[] arr = {1, 2, 3};

            try {
                System.out.println(arr[5]);  // This will throw an ArrayIndexOutOfBoundsException
            } catch (ArrayIndexOutOfBoundsException e) {
                System.err.println("Inner Catch: " + e.getMessage());
            }

            System.out.println("This statement will be executed despite the inner exception.");
        } catch (Exception e) {
            System.err.println("Outer Catch: " + e.getMessage());
        }
    }
}
```

Here, the outer `catch` block handles any exceptions that occur within the nested `try` block.

## 6.6 Finally Block:

```
public class FinallyBlockExample {
    public static void main(String[] args) {
        try {
            // Code that may throw exceptions
            System.out.println("Try Block");
        } catch (Exception e) {
            System.err.println("Catch Block: " + e.getMessage());
        } finally {
            // Code in the finally block will always execute
            System.out.println("Finally Block");
        }
    }
}
```

The `finally` block is useful for cleanup operations or releasing resources, and it always gets executed whether an exception occurs or not.

## 6.7 User-Defined Exceptions:

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

```
public class UserDefinedExceptionExample {
    public static void main(String[] args) {
        try {
            throw new CustomException("This is a custom exception.");
        } catch (CustomException e) {
            System.err.println("Custom Exception Caught: " + e.getMessage());
        }
    }
}
```

Here, `CustomException` is a user-defined exception class that extends the `Exception` class. The program throws and catches an instance of this custom exception.

In conclusion, Java's exception handling mechanisms provide a robust framework for dealing with runtime errors and exceptional conditions. By using `try`, `catch`, `finally`, and user-defined exceptions, developers can create more reliable and fault-tolerant applications.

**Exception Handling in Java with More Examples:**

Let's delve deeper into Java's exception handling model with additional examples to illustrate various aspects of handling exceptions.

**1. Checked Exception:**
- Checked exceptions are those that the compiler forces you to handle. Let's consider an example using `FileReader`:

```
import java.io.FileReader;
import java.io.IOException;

public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            FileReader fileReader = new FileReader("example.txt");
            // Read from the file
            fileReader.close();
        } catch (IOException e) {
            System.err.println("IOException: " + e.getMessage());
        }
    }
}
```

In this example, we use `FileReader` which throws `IOException`. The `try` block attempts to read from the file, and the `catch` block handles any potential `IOException`.

**2. Unchecked Exception:**
- Unchecked exceptions, also known as runtime exceptions, are not enforced by the compiler. An example using `ArrayIndexOutOfBoundsException`:

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3};
        try {
            System.out.println(arr[5]);  // This will throw an ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println("ArrayIndexOutOfBoundsException: " + e.getMessage());
```

```
        }
    }
}
```

Here, the `try` block attempts to access an index that is out of bounds, resulting in an `ArrayIndexOutOfBoundsException`.

**3. Multiple Catch Clauses:**
- Handling multiple types of exceptions within the same `try` block:

```
public class MultipleCatchExample {
    public static void main(String[] args) {
        try {
            String str = null;
            System.out.println(str.length());  // This will throw a NullPointerException
        } catch (NullPointerException e) {
            System.err.println("NullPointerException: " + e.getMessage());
        } catch (Exception e) {
            System.err.println("Generic Exception: " + e.getMessage());
        }
    }
}
```

In this example, the first `catch` block handles `NullPointerException`, and the second one provides a more generic catch-all for any other exceptions.

**4. Nested Try Statements:**
- Demonstrating nested `try` blocks:

```
public class NestedTryExample {
    public static void main(String[] args) {
        try {
            int[] arr = {1, 2, 3};

            try {
                System.out.println(arr[5]);  // This will throw an ArrayIndexOutOfBoundsException
            } catch (ArrayIndexOutOfBoundsException e) {
                System.err.println("Inner Catch: " + e.getMessage());
            }

            System.out.println("This statement will be executed despite the inner exception.");
        } catch (Exception e) {
            System.err.println("Outer Catch: " + e.getMessage());
        }
    }
}
```

In this example, the outer `catch` block handles any exceptions that occur within the nested `try` block.

**5. Finally Block:**
- Using `finally` for cleanup operations:

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

```
        try {
            // Code that may throw exceptions
            System.out.println("Try Block");
        } catch (Exception e) {
            System.err.println("Catch Block: " + e.getMessage());
        } finally {
            // Code in the finally block will always execute
            System.out.println("Finally Block");
        }
    }
}
```

The `finally` block is executed whether an exception occurs or not, making it suitable for cleanup operations.

**6. User-Defined Exception:**

- Creating a custom exception class:

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

public class UserDefinedExceptionExample {
    public static void main(String[] args) {
        try {
            throw new CustomException("This is a custom exception.");
        } catch (CustomException e) {
            System.err.println("Custom Exception Caught: " + e.getMessage());
        }
    }
}
```

In this example, `CustomException` is a user-defined exception class extending the `Exception` class. The program throws and catches an instance of this custom exception.

**Conclusion**

In conclusion, these examples provide a comprehensive overview of Java's exception handling mechanisms, covering checked and unchecked exceptions, multiple catch clauses, nested try statements, the finally block, and user-defined exceptions. Effective use of exception handling contributes to the creation of robust and reliable Java applications.

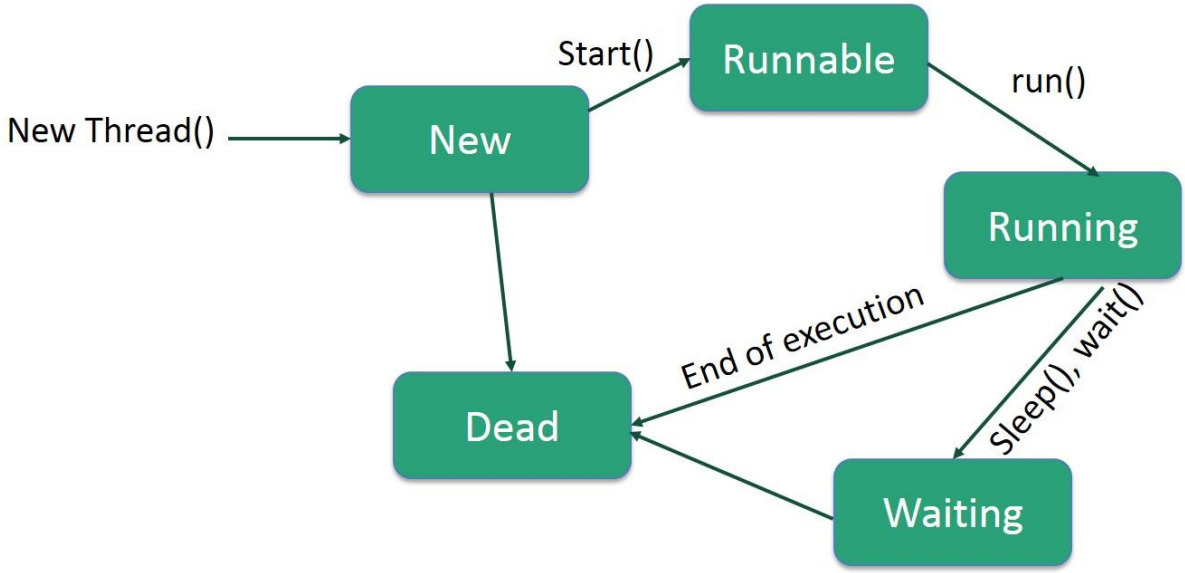
# M.Sc. (Computer Science)
# SEMESTER-3
# COURSE: Web Programming

| UNIT 7: MULTI-THREADED PROGRAMMING |
|---|
| 7.1 The Java Thread model |
| 7.2 The Thread class and Runnable interface |
| 7.3 Creating a Thread using Runnable Interface and extending Thread |
| 7.4 Creating Multiple Threads |
| 7.5 Thread Priorities |
| 7.6 Synchronizations: Methods, Statements, Inter Thread Communication, Deadlock, Suspending, Resuming and Stopping Threads. |

**Multi-threaded Programming in Java:**

Multi-threaded programming involves the concurrent execution of multiple threads within a program. In Java, threads are implemented through the `Thread` class and the `Runnable` interface. This approach allows for parallel execution of tasks, improving performance in scenarios where certain operations can be carried out simultaneously.

## 7.1 Java Thread Model:




Java's thread model is based on a two-tier architecture: the user-level threads (also known as application or green threads) and the kernel-level threads managed by the underlying operating system. Java threads are lightweight, and the Java Virtual Machine (JVM) takes care of mapping user-level threads to kernel-level threads.

## 7.2 Thread Class and Runnable Interface:

In Java, you can create threads by extending the `Thread` class or implementing the `Runnable` interface. The `Thread` class provides the basic functionality for creating and managing threads, while the `Runnable` interface allows you to define the code that will be executed by the thread.

# 7.3 Creating a Thread using Runnable Interface:

Here's an example of creating a thread by implementing the `Runnable` interface:

```
class MyRunnable implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getId() + " Value " + i);
        }
    }
}

public class RunnableExample {
    public static void main(String args[]) {
        Thread t1 = new Thread(new MyRunnable());
        Thread t2 = new Thread(new MyRunnable());

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

In this example, the `MyRunnable` class implements the `Runnable` interface, and the `run` method contains the code that will be executed when the thread runs. We create two threads (`t1` and `t2`) by passing instances of `MyRunnable` to the `Thread` constructor. The `start()` method is called to initiate the execution of the threads.

**Creating a Thread by Extending Thread Class:**

Here's an example of creating a thread by extending the `Thread` class:

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getId() + " Value " + i);
        }
    }
}

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

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

In this example, the `MyThread` class extends the `Thread` class, and the `run` method contains the code to be executed by the thread. We create two threads (`t1` and `t2`) by instantiating `MyThread` and then calling the `start()` method to begin the thread execution.

**Thread Synchronization:**

When multiple threads access shared resources concurrently, it can lead to data inconsistency or unexpected results. Java provides synchronization mechanisms, such as the `synchronized` keyword and locks, to ensure that only one thread can access critical sections of code at a time.

Here's a simple example using synchronization to avoid data inconsistency:

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

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

    public int getCount() {
        return count;
    }
}

class MyRunnable implements Runnable {
    private Counter counter;

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

    public void run() {
        for (int i = 0; i < 10000; i++) {
            counter.increment();
        }
    }
}

public class SynchronizationExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Thread t1 = new Thread(new MyRunnable(counter));
        Thread t2 = new Thread(new MyRunnable(counter));

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

        t1.join();
        t2.join();

        System.out.println("Final Count: " + counter.getCount());
    }
}
```

In this example, the `Counter` class has a synchronized `increment` method to ensure that only one thread can increment the count at a time. The `SynchronizationExample` class creates two threads that share the same `Counter` instance.

These examples provide a foundational understanding of multi-threaded programming in Java, covering the creation of threads using both the `Runnable` interface and the `Thread` class, as well as

addressing synchronization concerns when multiple threads access shared resources.

## 7.4 Creating Multiple Threads:

Creating multiple threads in Java can be achieved by either extending the `Thread` class or implementing the `Runnable` interface.

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

public class MultipleThreadsExample {
    public static void main(String[] args) {
        MyThread thread1 = new MyThread();
        MyThread thread2 = new MyThread();

        thread1.start();
        thread2.start();
    }
}
```

In this example, two threads (`thread1` and `thread2`) are created by extending the `Thread` class, and each thread runs independently when started.

## 7.5 Thread Priorities:

Java threads can have priorities ranging from 1 to 10, where 1 is the lowest priority and 10 is the highest. The default priority is 5.

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

public class ThreadPriorityExample {
    public static void main(String[] args) {
        PriorityThread thread1 = new PriorityThread();
        PriorityThread thread2 = new PriorityThread();

        thread1.setPriority(Thread.MIN_PRIORITY);  // Set minimum priority (1)
        thread2.setPriority(Thread.MAX_PRIORITY);  // Set maximum priority (10)

        thread1.start();
        thread2.start();
    }
}
```

Here, `setPriority` is used to set the priority of each thread.

## 7.6 Synchronization:

Synchronization is crucial to avoid data inconsistency when multiple threads access shared resources.

It can be achieved using the `synchronized` keyword.

**- Synchronized method:**

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

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

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

**- Synchronized block:**

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

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

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

Both examples ensure that only one thread can increment the count at a time.

**Inter-Thread Communication:**
Inter-thread communication is the mechanism by which threads can communicate with each other. `wait()`, `notify()`, and `notifyAll()` methods are used for this purpose.

```
class Message {
   private String message;

   public synchronized String getMessage() {
      return message;
   }

   public synchronized void setMessage(String message) {
      this.message = message;
      notify();  // Notify waiting threads
   }
}

class Producer extends Thread {
   private Message message;

   public Producer(Message message) {
```

```
        this.message = message;
    }

    public void run() {
        synchronized (message) {
            message.setMessage("Hello from Producer!");
        }
    }
}

class Consumer extends Thread {
    private Message message;

    public Consumer(Message message) {
        this.message = message;
    }

    public void run() {
        synchronized (message) {
            while (message.getMessage() == null) {
                try {
                    message.wait();  // Wait for notification
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("Message received: " + message.getMessage());
        }
    }
}

public class InterThreadCommunicationExample {
    public static void main(String[] args) {
        Message message = new Message();
        Producer producer = new Producer(message);
        Consumer consumer = new Consumer(message);

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

Here, `Producer` sets a message, and `Consumer` waits until a message is available.

**Deadlock:**
A deadlock occurs when two or more threads are blocked forever, each waiting for the other to release a lock.

```
public class DeadlockExample {
    static class Resource {
        synchronized void method1(Resource resource) {
            resource.method2(this);
        }

        synchronized void method2(Resource resource) {
```

```
            // Some operations
        }
    }

    public static void main(String[] args) {
        Resource resource1 = new Resource();
        Resource resource2 = new Resource();

        new Thread(() -> resource1.method1(resource2)).start();
        new Thread(() -> resource2.method1(resource1)).start();
    }
}
```

In this example, two threads each call a method that synchronizes on the other's instance, leading to a deadlock.

**Suspending, Resuming, and Stopping Threads:**
The `suspend()`, `resume()`, and `stop()` methods are deprecated due to potential issues. Instead, it is recommended to use flags or other mechanisms to control thread execution.

```
class MyThread extends Thread {
    private volatile boolean suspended = false;

    public void run() {
        while (true) {
            if (!suspended) {
                // Perform operations
            }
        }
    }

    public void suspendThread() {
        suspended = true;
    }

    public void resumeThread() {
        suspended = false;
    }

    public void stopThread() {
        // Perform cleanup operations
        interrupt();
    }
}
```

In this example, a `volatile` flag is used to suspend and resume the thread. The `stopThread` method interrupts the thread, allowing it to perform cleanup operations.

## Conclusion

These examples cover creating multiple threads, setting thread priorities, synchronization, inter-thread communication, handling deadlocks, and managing thread execution. Keep in mind that proper synchronization and coordination are crucial for the correctness of multi-threaded programs.

# M.Sc. (Computer Science)
# SEMESTER-3
# COURSE: Web Programming

| UNIT 8: APPLET PROGRAMMING |
|---|
| 8.1 Introduction |
| 8.2 Types of applet |
| 8.3 Life Cycle |
| 8.4 Incorporating an applet into web page using Applet Tag |
| 8.5 Running applets |
| 8.6 Using Graphics class and its methods to draw lines, rectangles, circles, ellipses, arcs and polygons |

## 8.1 Introduction:

An applet is a small Java program that runs within a web browser. It is a client-side technology used to enhance the functionality of web pages. Applets are written in Java and are typically embedded in HTML pages.

## 8.2 Types of Applets:

There are two types of applets in Java:

**1. Applet Based on Applet Class (Heavyweight Applet):**

- This type of applet extends the `java.applet.Applet` class.
- It provides a graphical user interface.
- These applets are heavyweight because they use the native GUI components of the operating system.

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorldApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello, World!", 20, 20);
    }
}
```

**2. Applet Based on JApplet Class (Lightweight Applet):**

- This type of applet extends the `javax.swing.JApplet` class.
- It provides a more modern and versatile GUI.
- These applets are lightweight because they use Swing components.

```
import javax.swing.JApplet;
import java.awt.Graphics;

public class HelloWorldSwingApplet extends JApplet {
    public void paint(Graphics g) {
        g.drawString("Hello, World!", 20, 20);
    }
}
```

# 8.3 Life Cycle of an Applet:

The life cycle of an applet consists of the following methods:

1. `init()`: This method is called when the applet is first loaded. It is used for one-time initialization.

2. `start()`: This method is called after the `init()` method. It is invoked each time the user returns to the applet.

3. `stop()`: This method is called when the user leaves the page containing the applet. It can be used to stop ongoing processes.

4. `destroy()`: This method is called when the browser shuts down or the applet is removed from the page. It is used for cleanup operations.

5. `paint(Graphics g)`: This method is called whenever the applet needs to redraw its contents. It is frequently used to display graphics.


# 8.4 Incorporating an Applet into a Web Page Using the Applet Tag:

To include an applet in an HTML page, you use the `<applet>` tag. Here's an example:

```
<html>
  <body>
    <applet code="HelloWorldApplet.class" width="300" height="300">
    </applet>
  </body>
</html>
```

In this example, the `code` attribute specifies the name of the compiled applet class file, and the `width` and `height` attributes determine the dimensions of the applet.

# 8.5 Running Applets:

To run an applet, you can use a web browser that supports Java applets, such as Internet Explorer, Firefox, or Safari. Ensure that Java is installed on the system, and the browser's Java plugin is enabled.

**1. Compile the Applet Class:**
- Use the `javac` command to compile the Java source file:

*javac HelloWorldApplet.java*

**2. Create an HTML Page:**
- Create an HTML file that includes the `<applet>` tag.

**3. Open the HTML Page in a Browser:**
- Open the HTML file using a web browser.

The browser will load the applet and display it within the specified dimensions.

It's important to note that modern web standards have shifted away from using Java applets in favor of other technologies like JavaScript and HTML5. Applets may not be supported in some browsers due to security concerns, and alternative technologies are recommended for web development.

**More Explanation:**

The life cycle of an applet is managed by the browser or applet viewer. Here's a more detailed breakdown of each method in the life cycle:

1. `init()`:
- The `init()` method is called when the applet is first loaded. It is used for one-time initialization tasks, such as setting up GUI components, loading resources, or establishing connections.
- Example:

```
public void init() {
    // Initialization code here
}
```

2. `start()`:
- The `start()` method is called after the `init()` method. It is invoked each time the user returns to the applet or when the browser revisits the applet's page. This method is often used to start threads or animations.
- Example:

```
public void start() {
    // Start or resume any ongoing processes
}
```

3. `stop()`:
- The `stop()` method is called when the user leaves the page containing the applet. It is used to stop ongoing processes or animations to conserve system resources.
- Example:

```
public void stop() {
```

```
    // Stop ongoing processes or animations
  }
```
4. `destroy()`:
   - The `destroy()` method is called when the browser shuts down or the applet is removed from the page. It is used for cleanup operations, such as closing open resources or releasing memory.
   - Example:

```
  public void destroy() {
    // Cleanup operations
  }
```

5. `paint(Graphics g)`:
   - The `paint(Graphics g)` method is called whenever the applet needs to redraw its contents. This method is often used to display graphics or update the applet's visual representation.
   - Example:

```
  public void paint(Graphics g) {
    // Draw graphics on the applet's canvas
  }
```

**Incorporating an Applet into a Web Page Using the `<applet>` Tag:**

To include an applet in an HTML page, you use the `<applet>` tag. The following attributes are commonly used:

- `code`: Specifies the name of the compiled applet class file.
- `width` and `height`: Determine the dimensions of the applet.

Example HTML code:

```
<html>
  <body>
    <applet code="HelloWorldApplet.class" width="300" height="300">
    </applet>
  </body>
</html>
```

Here, the applet class file is specified as "HelloWorldApplet.class," and the applet is given a width and height of 300 pixels each.

**Running Applets:**

**1. Compile the Applet Class:**
   - Use the `javac` command to compile the Java source file:

```
javac HelloWorldApplet.java
```

**2. Create an HTML Page:**
   - Create an HTML file that includes the `<applet>` tag. Save the HTML file in the same directory as the compiled applet class.

   Example HTML file ("index.html"):

```
<html>
```

```
    <body>
        <applet code="HelloWorldApplet.class" width="300" height="300">
        </applet>
    </body>
</html>
```

**3. Open the HTML Page in a Browser:**

- Open the HTML file using a web browser that supports Java applets. Ensure that Java is installed on the system, and the browser's Java plugin is enabled.

The browser will load the applet and display it within the specified dimensions.

It's important to note that Java applets have fallen out of favor in modern web development due to security concerns, and major browsers have deprecated or removed support for Java plugins. As an alternative, technologies like JavaScript and HTML5 are commonly used for client-side web development.

# 8.6 Using Graphics class and its methods to draw lines, rectangles, circles, ellipses, arcs and polygons:

To draw lines, rectangles, circles, ellipses, arcs, and polygons in Java, you can use the `Graphics` class, which is part of the Abstract Window Toolkit (AWT) package. Below are examples demonstrating how to use various methods of the `Graphics` class to draw these shapes. For simplicity, I'll provide standalone examples for each shape.

**1. Drawing Lines:**

```
import java.awt.Graphics;
import javax.swing.JFrame;
import javax.swing.JPanel;

class LineDrawing extends JPanel {
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawLine(10, 10, 150, 150);
    }
}

public class LineExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Line Example");
        frame.setSize(200, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(new LineDrawing());
        frame.setVisible(true);
    }
}
```

**2. Drawing Rectangles:**

```
import java.awt.Graphics;
import javax.swing.JFrame;
import javax.swing.JPanel;

class RectangleDrawing extends JPanel {
```

```
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawRect(20, 20, 120, 80);
    }
}

public class RectangleExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Rectangle Example");
        frame.setSize(200, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(new RectangleDrawing());
        frame.setVisible(true);
    }
}
```

**3. Drawing Circles:**

```
import java.awt.Graphics;
import javax.swing.JFrame;
import javax.swing.JPanel;

class CircleDrawing extends JPanel {
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawOval(20, 20, 80, 80);
    }
}

public class CircleExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Circle Example");
        frame.setSize(200, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(new CircleDrawing());
        frame.setVisible(true);
    }
}
```

**4. Drawing Ellipses:**

```
import java.awt.Graphics;
import javax.swing.JFrame;
import javax.swing.JPanel;

class EllipseDrawing extends JPanel {
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawOval(20, 20, 120, 80);
    }
}
```

```
public class EllipseExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Ellipse Example");
        frame.setSize(200, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(new EllipseDrawing());
        frame.setVisible(true);
    }
}
```

**5. Drawing Arcs:**

```
import java.awt.Graphics;
import javax.swing.JFrame;
import javax.swing.JPanel;

class ArcDrawing extends JPanel {
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawArc(20, 20, 100, 100, 45, 90);
    }
}

public class ArcExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Arc Example");
        frame.setSize(200, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(new ArcDrawing());
        frame.setVisible(true);
    }
}
```

**6. Drawing Polygons:**

```
import java.awt.Graphics;
import java.awt.Polygon;
import javax.swing.JFrame;
import javax.swing.JPanel;

class PolygonDrawing extends JPanel {
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        int[] xPoints = {20, 80, 120, 60};
        int[] yPoints = {40, 20, 80, 120};
        int nPoints = 4;
        Polygon polygon = new Polygon(xPoints, yPoints, nPoints);
        g.drawPolygon(polygon);
    }
}

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

```
        JFrame frame = new JFrame("Polygon Example");
        frame.setSize(200, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(new PolygonDrawing());
        frame.setVisible(true);
    }
}
```

These examples provide a basic introduction to drawing lines, rectangles, circles, ellipses, arcs, and polygons using the `Graphics` class in Java. You can further customize and enhance these examples based on your specific requirements.

# Theoretical Assignments

1. What are the key characteristics of the Java programming language discussed in Unit 1?
2. Explain the role of the Java development kit and Java runtime environment in Java programming.
3. Differentiate between Java and C++ in terms of their features and functionalities.
4. Discuss the concept of Byte Code and its significance in the Java Virtual Machine.
5. How does the Java language handle constants, variables, and data types? Provide examples.
6. Explore the various operators and expressions in Java, highlighting their usage and importance.
7. What are the control structures in Java, and how do they contribute to program flow control?
8. Explain the process of defining a class and creating objects in Java, as covered in Unit II.
9. Discuss the role of constructors in Java, emphasizing their importance in object initialization.
10. What is Garbage Collection in Java, and how does it impact memory management?
11. Compare and contrast different types of inheritance discussed in Unit III.
12. Demonstrate the use of the super keyword to call superclass constructors, creating a multilevel hierarchy in Java.
13. Explore the concept of method overriding and dynamic method dispatch in Java.
14. How can abstract classes be used in Java, and what is the significance of the final keyword?
15. Provide an overview of I/O Basics, including streams and predefined streams in Java.
16. How is console input read and console output written in Java?
17. Explain the concepts of one-dimensional and two-dimensional arrays in Java.
18. Discuss the String handling mechanisms using String and StringBuffer classes.
19. Explore the different types of packages in Java and discuss the process of defining and importing packages.
20. Describe the principles of exception handling in Java, including types of exceptions, try-catch blocks, and user-defined exceptions.

# Practical Assignments

**1. Java Programming (Unit II):**

- Write a Java program to demonstrate the use of variables, constants, and different data types.
- Implement a Java program that uses control structures (if-else, switch) for decision-making.
- Develop a Java program that involves the use of loops for repetitive tasks.

**2. Inheritance (Unit III):**

- Create a Java program illustrating single inheritance and multiple inheritance using classes.
- Implement a multilevel hierarchy in Java, showcasing the use of the super keyword.
- Develop a Java program that uses abstract classes for achieving abstraction and code reusability.

**3. Arrays and Strings (Unit IV):**

- Write a Java program to manipulate a one-dimensional array.
- Implement a program that uses a two-dimensional array to store and manipulate data.
- Explore String handling in Java by writing a program that uses String and StringBuffer classes.

**4. Exception Handling (Unit VI):**

- Develop a Java program that deliberately generates and handles exceptions using try-catch blocks.
- Implement a program that demonstrates the use of multiple try-catch clauses for different exceptions.
- Create a custom Java exception and use it in a program with appropriate handling.

**5. Multi-threaded Programming (Unit VII):**

- Write a Java program that illustrates the creation of threads using both the Thread class and Runnable interface.
- Implement a program that demonstrates thread synchronization using methods and statements.
- Create a Java program showcasing inter-thread communication to synchronize the execution of threads.

**6. Applet Programming (Unit VIII):**

- Develop a basic Java applet that draws simple shapes (lines, rectangles, circles) using the Graphics class.
- Incorporate an applet into an HTML page using the <applet> tag and run it in a web browser.
- Write a Java applet that responds to user events, such as mouse clicks or key presses.

These programming exercises cover various concepts from Java programming, ensuring a practical understanding of the topics discussed in the respective units.