Modern Java - Volume I

Navid Mohaghegh

Modern Java - Volume I

Navid Mohaghegh

Contents

Preface

The world of programming is dynamic, constantly evolving to meet the challenges of emerging technologies. Java, with its adaptability, robustness, and ever-expanding ecosystem, stands at the forefront of this evolution. From its inception as a language designed to bridge the gap between platform independence and developer productivity, Java has grown into a comprehensive toolset for building everything from simple desktop applications to complex, distributed systems.

Why This Book?

In a world brimming with resources on programming, this book serves a dual purpose: as a gateway for beginners embarking on their Java journey and as a practical reference for seasoned developers looking to stay updated with the language's latest features and best practices. By combining foundational concepts with advanced techniques, this book offers a holistic approach to mastering Java, catering to learners at all stages.

What Sets This Book Apart?

- **Structured Learning Path**: The chapters are designed to progress logically, starting with the basics and gradually moving to advanced topics like concurrency, networking, and design patterns.
- **Comprehensive Coverage**: From setting up your development environment to understanding the intricacies of JVM internals, this book leaves no stone unturned.
- **Practical Examples**: Real-world scenarios and code snippets bridge the gap between theory and practice, ensuring readers can apply their knowledge effectively.
- **Focus on Modern Java**: Emphasis on the latest Java versions and features ensures readers stay current in a rapidly changing industry.

How to Use This Book

This book is divided into ten major sections, each addressing key aspects of Java programming:

1. **Introduction and Essentials**: Lay the foundation by exploring the history of programming languages, setting up the Java environment, and understanding the anatomy of a basic Java program.
2. **Basic Language Concepts**: Dive into syntax, variables, data types, and control flow, building a strong base for more advanced topics.
3. **Intermediate Concepts**: Explore object-oriented programming, classes, inheritance, polymorphism, and encapsulation.
4. **Java Collections and Maps**: Master data structures like lists, sets, and maps, and learn efficient ways to handle data.
5. **Java Standard Library and Utilities**: Familiarize yourself with essential utilities, file handling, and the java.time API.
6. **Concurrency**: Understand threading, synchronization, and the new virtual threads introduced in Project Loom.
7. **Networking**: Learn socket programming, HTTP communication, and practical applications in client-server models.
8. **Graphical User Interfaces**: Create interactive GUIs using Swing, with hands-on examples and best practices.
9. **Design Patterns**: Explore the most influential design patterns and their implementation in Java.
10. **Advanced Java Concepts**: Discover lambda expressions, streams, reflection, annotations, and modular programming.

Who Should Read This Book?

This book is ideal for:

- **Students**: Those beginning their journey into Java programming or seeking a structured resource for their coursework.
- **Hobbyists and Enthusiasts**: Individuals eager to explore Java's capabilities in building diverse applications.
- **Professional Developers**: Practitioners looking to deepen their understanding of modern Java features and apply best practices in their work.

Acknowledgments

This book is a culmination of years of programming experience, community contributions, and countless hours of research. I extend my gratitude to the Java developer community, educators, and open-source contributors whose work inspires millions globally. Special thanks to my readers—you are the driving force behind this endeavor.

As you navigate the chapters of this book, I hope it empowers you to create, innovate, and solve problems using the elegance and power of Java. Let this journey into Java not just expand your technical skillset but also inspire a deeper appreciation for the art of programming.

Navid Mohaghegh
December 2024

Chapter 1

Introductions and Essentials

1.1 Introduction to Programming and Java

Programming is the process of designing and writing instructions that a computer can execute to perform specific tasks. These instructions are expressed using a programming language, which serves as a medium between human logic and machine operation. Learning programming equips individuals with the ability to analyze problems, design algorithms, and develop solutions through software, enabling automation and innovation across various industries.

The history of programming languages has evolved significantly, with languages like Assembly, C, and Python shaping modern development. Among these, Java holds a unique position due to its simplicity, robustness, and platform independence. Java was developed in the mid-1990s by Sun Microsystems and has grown into one of the most widely used programming languages in the world. It combines the best features of earlier languages, such as C++ and Smalltalk, while addressing many of their limitations.

At its core, Java is an object-oriented programming (OOP) language. Object-oriented programming emphasizes modular design, where real-world entities are modeled as objects. These objects encapsulate both data (attributes) and behaviors (methods), promoting code reuse, scalability, and maintainability. This approach helps simplify the design of complex systems, making Java suitable for building enterprise applications, games, mobile software, and many more.

One of Java's defining characteristics is its platform independence, often summarized by the phrase "Write Once, Run Anywhere." Java programs are compiled into an intermediate format called bytecode, which runs on the Java Virtual Machine (JVM). This virtual machine abstracts the underlying hardware and operating system, ensuring that Java applications can execute seamlessly across platforms like Windows, Linux, and macOS.

The Java programming language prioritizes simplicity and ease of use, especially for beginners. Its syntax is designed to be clear and similar to C/C++, making it accessible to developers transitioning from other languages. At the same time, Java removes complex and error-prone features such as explicit memory management, allowing programmers to focus on problem-solving rather than system-level details.

Another significant feature of Java is its automatic memory management through garbage collection. In languages like C and C++, developers must manually allocate and deallocate memory, which can lead to errors like memory leaks or segmentation faults. Java eliminates these issues by automatically reclaiming unused memory, resulting in more reliable and stable applications.

Java supports multi-threading, a feature that enables applications to execute

multiple tasks simultaneously. Threads are lightweight processes that run independently, allowing developers to design programs that efficiently utilize modern multi-core processors. Java provides built-in thread management capabilities, making it a preferred choice for building responsive and high-performance applications.

Exception handling is another area where Java excels. Errors in a program can cause it to crash or behave unpredictably. Java introduces a robust exception-handling mechanism that ensures programs can detect, manage, and recover from runtime errors gracefully. The use of try, catch, finally, and throw constructs provides developers with tools to build fault-tolerant software.

In addition to being feature-rich, Java is known for its security. Java's runtime environment, the JVM, incorporates security measures like bytecode verification and sandboxing. These features protect systems from malicious code and unauthorized access, which is particularly important for applications distributed over the internet.

Java's vast standard library, also known as the Java Development Kit (JDK), is another strength of the language. The JDK includes thousands of classes and methods for tasks such as file manipulation, network communication, database access, and graphical user interface (GUI) creation. With these tools, developers can build powerful and feature-rich applications quickly and efficiently.

To simplify GUI development, Java provides the Swing and JavaFX libraries. Swing allows developers to create lightweight graphical interfaces, while JavaFX adds advanced capabilities like multimedia, animations, and styling with cascading style sheets. These tools make Java suitable for building modern desktop applications with attractive user interfaces.

Java is widely used in web development. Technologies such as Java Servlets, JavaServer Pages (JSP), and frameworks like Spring and Struts enable developers to build scalable, secure, and dynamic web applications. Java's ability to handle HTTP requests, sessions, and server-side logic makes it a suitable tool for enterprise-level web solutions.

The popularity of Android development has further cemented Java's importance. Android applications are written primarily in Java-flavored languages, leveraging the Android SDK and tools like Android Studio. Java's versatility and performance make it ideal for building mobile applications that run on billions of devices worldwide.

Another critical aspect of Java's ecosystem is its support for database connectivity through the Java Database Connectivity Application Programming Interfaces (JDBC APIs). JDBC allows developers to interact with various relational databases like MySQL, PostgreSQL, and Oracle, enabling the storage, retrieval, and manipulation of structured data seamlessly.

Java also plays a significant role in enterprise software development. Technologies

like Enterprise JavaBeans (EJB) and platforms and reference architectures such as Java EE provide tools and frameworks for building large-scale, distributed systems. Java's scalability and reliability make it the preferred choice for banking, insurance, and e-commerce applications.

In recent years, Java has evolved to include features that make it more modern and competitive. With versions like JDK 11 through JDK 23, Java has introduced enhancements such as the Java Platform Module System (JPMS), local variable type inference (var), switch expressions, records, and pattern matching. These features simplify coding, improve performance, and bring Plain Old Java Objects (POJO) closer to the other Java-flavored languages like Kotlin and Scala.

Java's support for functional programming has grown with the introduction of lambda expressions and the Stream API in Java 8. These tools enable developers to write more concise, readable, and efficient code, particularly when processing collections of data.

The Java community is another reason for its enduring success. With millions of developers, open-source libraries, and online resources, Java offers unmatched support for beginners and professionals alike. Communities like Stack Overflow, GitHub, and the Oracle Java forums provide solutions to common programming challenges.

Java's relevance in the era of cloud computing and big data is also noteworthy. Frameworks such as Apache Hadoop, Spark, and Kafka are built on Java, enabling scalable data processing and analytics. Java's compatibility with cloud platforms like AWS, Azure, and Google Cloud ensures its role in building distributed, cloud-native applications.

Learning Java is an excellent starting point for aspiring programmers because it introduces fundamental programming concepts such as variables, data types, loops, conditionals, and methods. Beginners can focus on mastering these principles without being overwhelmed by language complexities.

The transition from basic to advanced topics in Java is seamless. Once beginners understand core programming concepts, they can explore advanced features like object-oriented design, generics, and multithreading. Java's comprehensive tools and libraries allow students to gradually build projects of increasing complexity.

Java programming also emphasizes code organization and best practices. With features like packages, interfaces, and design patterns, developers can write clean, modular, and reusable code. Tools such as Maven and Gradle further simplify project management, builds, and dependency handling.

Java continues to be an important part in programming education, offering educators an ideal platform to introduce students to software development. Its clarity, strict typing, and robust development tools create an environment where

learners can grasp fundamental concepts with confidence. As students master Java's principles of object-oriented design and systematic problem-solving, they develop transferable skills that seamlessly apply across the programming landscape.

In the professional world, Java's platform independence and robust ecosystem enable developers to build everything from mobile apps to enterprise cloud-native systems. Its security features and continuous evolution make it well-suited for modern challenges like cloud computing and microservices.

This book takes a practical approach, offering a concise yet comprehensive exploration of modern Java's powerful features. Through hands-on examples and real-world applications, readers will discover how Java's versatility can open doors to diverse career opportunities in technology, whether in software engineering, cloud infrastructure, or secure application development.

1.1.1 Brief History and Evolution of Programming Languages

The evolution of programming languages is a fascinating journey that highlights humanity's ingenuity in improving how we interact with computers. From low-level machine code to modern high-level languages like Java, this progression has been driven by the need for better performance, productivity, and ease of use. Here we quickly explore the key stages in the evolution of programming languages, their characteristics, and how they have shaped the modern software development landscape.

1.1.1.0.1 The Dawn of Machine Code (1940s-1950s) The earliest computers were programmed using **machine code**, which consists of binary instructions (0s and 1s) understood directly by the computer's hardware. These instructions represented fundamental operations like loading data, performing arithmetic, and storing results. For example, a typical machine instructions (in hexadecimal representation) could look like:

$$B80100000083C002A300040000$$

This code might represent an operations to add two values, but it is cryptic and error-prone. Programming at this level required intimate knowledge of hardware architecture, making development slow and tedious.

1.1.1.0.2 Assembly Language (1950s) To make programming more human-readable, **assembly languages** were introduced. Assembly languages replaced binary

machine instructions with symbolic representations called mnemonics. Programmers could write instructions like:

```
B8 01 00 00 00   mov eax, 1          ; Load 1 into eax
83 C0 02         add eax, 2          ; Add 2 to eax
A3 00 04 00 00   mov [0x400], eax    ; Store result at mem 1024
```

Assemblers translated this symbolic code into machine code. While assembly languages simplified programming, they remained hardware-specific, requiring programmers to rewrite code for different machines.

1.1.1.0.3 The Birth of High-Level Languages (1950s-1960s) In the late 1950s, the first **high-level programming languages** emerged, abstracting hardware details to enable human-friendly code. These languages introduced constructs for loops, conditionals, and subroutines, allowing programmers to focus on problem-solving rather than hardware intricacies. **Fortran** (Formula Translation), developed in 1957, was one of the first high-level languages, designed for scientific and mathematical computing:

```
DO 10 I = 1, 100 A(I) = B(I) + C(I) 10 CONTINUE
```

which essentially takes the first 100 elements of array B and the first 100 elements of array C, add them together element by element, and store the results in the first 100 elements of array A.

1.1.1.0.4 COBOL and Business Applications (1960s) In 1960, **COBOL** (Common Business Oriented Language) was developed to handle business applications. Its syntax was designed to be close to English, making it easier for non-programmers to understand.

```
PERFORM UNTIL COUNTER = 100 ADD 1 TO COUNTER DISPLAY "Processing Record" END-PERF
```

COBOL revolutionized business computing, remaining in use for decades, especially in financial systems.

1.1.1.0.5 Structured Programming and C (1970s) By the 1970s, the focus shifted to improving program structure and maintainability. **Structured programming** introduced clear constructs like loops, conditionals, and functions, avoiding ”spaghetti code” caused by unrestricted `goto` statements. **C**, developed by Dennis Ritchie in 1972, became the norm of structured programming. It combined performance with portability and simplicity:


```c
#include <stdio.h>
int main() {
  int i;
  for (i = 0; i < 10; i++) {
    printf("Hello, World!\n");
  }
  return 0;
}
```

C's flexibility allowed it to be used for system programming, operating systems, and application development, laying the groundwork for modern programming. Many modern operating systems that we are using right now are written in C language!

1.1.1.0.6 The Object-Oriented Paradigm (1980s)

The increasing complexity of software demanded a new way of organizing code. The **object-oriented programming (OOP)** paradigm emerged, encapsulating data and behavior into objects. **Smalltalk**, developed in the 1970s, was the first pure OOP language. C++ extended C with object-oriented features in the mid-1980s. Programs could now define classes and objects, improving code reuse and maintainability:

```cpp
class Car {
        public:
        void drive() {
                cout << "Driving the car";
        }
};

int main() {
        Car myCar;
        myCar.drive();
        return 0;
}
```

1.1.1.0.7 Java and Platform Independence (1990s)

Java, introduced in 1995, revolutionized programming with its "Write Once, Run Anywhere" philosophy. Java programs are compiled into bytecode that runs on the Java Virtual Machine (JVM), ensuring platform independence. Java also popularized garbage collection and safe memory management.

```
public class HelloWorld { public static void main(String[] args) { System.out.println('
```


1.1.1.0.8 Scripting Languages and Web Development (1990s-2000s) With the rise of the internet, lightweight **scripting languages** such as JavaScript, Perl, PHP, and Python became popular for building dynamic web applications. These languages emphasized rapid development and ease of use:

```
// JavaScript
function greet() {
        console.log("Hello, World!");
}

greet();
```

1.1.1.0.9 Functional Programming Renaissance (2000s) Functional programming, an older paradigm from the 1950s (e.g., Lisp), saw a resurgence in the 2000s with languages like Haskell, Scala, and features added to Java (e.g., lambdas in Java 8). Functional programming emphasizes immutability, higher-order functions, and declarative coding:

```
// Java Lambdas
List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
numbers.stream()
.map(x -> x * 2)
.forEach(System.out::println);
```

1.1.1.0.10 Modern Languages and JVM Ecosystem (2010s-Present) Modern programming languages like Kotlin, Go, and Rust prioritize safety, simplicity, and concurrency. Kotlin, for example, has become a popular JVM-based language for Android development:

```
// Kotlin
fun main() {
        println("Hello, Kotlin!")
}
```

Kotlin combines Java's strengths with concise syntax, null safety, and coroutines for asynchronous programming. Kotlin coroutines provide a sophisticated yet approachable solution for asynchronous and non-blocking programming. Unlike traditional threads, coroutines are lightweight and can be suspended and resumed without blocking the underlying thread. This makes them particularly efficient when dealing with operations that might take time, such as network calls, database operations, or file I/O. The 'suspend' keyword marks functions that can be paused

and resumed, allowing other coroutines to run on the same thread during the pause. Coroutines operate within CoroutineScopes and are governed by CoroutineContext, which defines their behavior and lifecycle. Kotlin provides several coroutine builders like launch, async, and runBlocking, each serving different purposes in concurrent programming. For example, 'launch' starts a new coroutine without blocking the current thread, while 'async' returns a Deferred result that can be awaited. This structured concurrency ensures that when a coroutine is cancelled, all its child coroutines are cancelled too, preventing memory leaks and simplifying error handling. The combination of suspend functions, coroutine scopes, and dispatchers makes it easier to write concurrent code that is both efficient and maintainable.

Similar aproaches are taken by other languages as well. For instance, Go lang approaches the concurrency centers on goroutines and channels, providing a simple yet powerful model for concurrent programming. Goroutines are lightweight threads managed by the Go runtime, allowing developers to run functions concurrently with minimal overhead. Unlike traditional threads that might consume megabytes of memory, goroutines start with just a few kilobytes and can grow or shrink as needed. The Go runtime automatically handles the scheduling of goroutines across available OS threads, making efficient use of system resources. The simple syntax go function() spawns a new goroutine, making concurrent programming remarkably accessible.

Channels complement goroutines by providing a safe way for goroutines to communicate and synchronize their execution. Following Go's philosophy of "Don't communicate by sharing memory; share memory by communicating," channels act as typed conduits for sending and receiving data between goroutines. This approach naturally prevents common concurrency issues like race conditions and deadlocks. The select statement enhances this model by allowing goroutines to wait on multiple channel operations simultaneously, while context packages provide elegant ways to handle timeouts, cancellation, and deadline propagation across API boundaries. Combined with Go's built-in support for synchronization primitives like mutexes and wait groups, this concurrency model enables developers to build robust, concurrent applications with relatively simple code.

1.1.1.0.11 Convergence of Paradigms and Language Features Modern programming languages are witnessing a remarkable convergence of programming paradigms and features, with Java demonstrating significant evolution to compete with newer languages. Java's Project Loom introduces virtual threads, offering lightweight concurrency similar to Go's goroutines and Kotlin's coroutines. Virtual threads can scale to millions of instances while maintaining the familiar thread programming model, allowing developers to write highly concurrent applications with

minimal overhead and complexity.

Java's functional programming capabilities have matured substantially. The Stream API, introduced in Java 8 and enhanced in subsequent releases, provides powerful data processing operations comparable to Kotlin's sequence operations. Pattern matching for switch expressions and records (Java 14+) bring concise, expressive syntax for data manipulation, rivaling Kotlin's smart casts and data classes. Records eliminate boilerplate code for data carriers, similar to Kotlin's data classes and Go's structs, while maintaining Java's strong type safety.

In the realm of asynchronous programming, Java's CompletableFuture API offers sophisticated composition and error handling capabilities. While different from Kotlin's coroutines or Go's goroutines, it provides a robust foundation for building reactive applications. The introduction of the Reactive Streams API and frameworks like Project Reactor further enhances Java's capabilities in handling asynchronous data streams and back-pressure, essential features for modern microservices architectures.

Java continues to evolve with Project Amber, introducing sealed classes, pattern matching for instanceof, and text blocks, bringing feature parity with modern languages while maintaining backward compatibility. The language's rich ecosystem, comprehensive tooling, and enterprise-grade libraries ensure its relevance in contemporary software development, whether for cloud-native applications, microservices, or large-scale distributed systems.

1.1.1.0.12 The Future of Programming Languages The future of programming will continue to evolve toward increased abstraction, automation, and integration with technologies like artificial intelligence. Programming languages will likely emphasize:

- Concurrency and parallelism for multi-core processors.
- Safer memory management and compile-time verification.
- Declarative and domain-specific languages for specialized tasks.
- Integration with machine learning and big data ecosystems.

From machine code to modern languages like Java, programming has evolved to make software development faster, safer, and more efficient. Each stage of evolution has addressed challenges such as hardware dependency, software complexity, and developer productivity. Java, standing at the intersection of performance, safety, and modern features, continues to play an important role in the future of programming.

1.2 Java Development Environment Setup

Setting up the development environment is the first step to getting started with Java programming. This chapter guides you through downloading, installing, and configuring the necessary tools, including the Java Development Kit (JDK) and an Integrated Development Environment (IDE). It also covers environment variable configuration and installation verification to ensure your setup is ready for Java development.

1.2.1 Downloading, Installing, and Setting Up the Java Development Environment

The Java Development Kit (JDK) is a software package that provides the tools required to develop, compile, and execute Java programs. It includes:

- **Java Compiler** (`javac`): Converts Java source code into bytecode.
- **Java Runtime Environment (JRE)**: Executes compiled bytecode.
- **Development Tools**: Utilities like `javadoc`, `jar`, and `java`.

To download and install JDK:

1. Visit the official Oracle or OpenJDK website:
    - Oracle JDK: `https://www.oracle.com/java`
    - OpenJDK: `https://openjdk.org`
2. Choose the appropriate JDK version for your platform (e.g., JDK 21 or 23).
3. Download the installer (e.g., `.exe`, `.dmg`, or `.tar.gz`).

1.2.1.0.1 For Windows:

1. Run the downloaded installer.
2. Follow the installation wizard and note the installation path (e.g., `C:\Program Files\Java\jdk-21`).

1.2.1.0.2 For macOS:

1. Run the `.dmg` installer.
2. Complete the setup by following the prompts.

1.2.1.0.3 For Linux:

1. Extract the downloaded archive:

```
tar zxvf jdk-[version].tar.gz
sudo mv jdk-[version] /usr/local/
```

2. Alternatively, install via a package manager:

```
sudo apt update
sudo apt install openjdk-21-jdk
```

1.2.2 Installing IntelliJ IDE

An Integrated Development Environment (IDE) simplifies Java development by providing features like code completion, debugging, and build tools. IntelliJ IDEA is one of the most popular IDEs for Java. Here are the steps to download and install IntelliJ IDEA:

1. Visit the official JetBrains website: `https://www.jetbrains.com/idea/`.
2. Download the appropriate installer for your operating system.
3. Choose between:
    - **Community Edition** (Free): Suitable for most Java applications.
    - **Ultimate Edition** (Paid): Includes advanced features for enterprise development.

1.2.3 Creating Your First Java Project in IntelliJ IDEA

Writing a "Hello, World!" program is the traditional starting point for learning any programming language. It introduces basic concepts such as the program structure, syntax, and tools required for Java development.

A Java program is written in a plain text file with the extension `.java`. The file name must match the name of the public class that contains the `main` method. Below is an example of creating a source file named `HelloWorld.java`:

1. Launch IntelliJ IDEA.
2. Select `New Project`.
3. Choose the JDK version and project type.
4. Create a file named `HelloWorld.java` and add the following code. Please note that Java is case-sensitive, and you must follow the letter cases exactly as shown here.

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

5. Click the **Run** button to execute the program.

1.3 Environment Variables Configuration

Operating System Environmental Variables allow the applications to locate the installed Java tools. The key variables for Java development are:

- **JAVA_HOME**: Points to the JDK installation directory.
- **PATH**: Specifies the directories containing executable files like `javac` and `java`.

1.3.1 Setting Up Environment Variables

1.3.1.0.1 For Windows:

1. Right-click on **This PC** or **Computer**, then select **Properties**.
2. Navigate to **Advanced system settings** → **Environment Variables**.
3. Add a new system variable:
   - `Name: JAVA_HOME`
   - `Value: C:\Program Files\Java\jdk-21`
4. Edit the `PATH` variable to include:

```
%JAVA_HOME%\bin
```


1.3.1.0.2 For macOS and Linux:

1. Open the shell configuration file (`/.zshrc` or `/.bashrc`).
2. Add the following lines:

```
export JAVA_HOME=/usr/local/jdk-21
export PATH=$JAVA_HOME/bin:$PATH
```

3. Save the file and apply the changes:

```
source ~/.zshrc
```

1.3.2 Installation Verification

Just to recap, below is a typical setup that one may do in MacOS:

```
# Change this to where you want to have your JDK-related files.
export TARGET_DIR=${HOME}/java

# In osX if you have Brew program, you can run:
#    brew install openjdk@23 gradle maven
# Once installed, check and change below paths as needed. They depend on the version being installed.
export BREW_JAVA=/opt/homebrew/Cellar/openjdk/23.0.1/libexec/openjdk.jdk/Contents/Home
export BREW_GRADLE=/opt/homebrew/Cellar/gradle/8.12/bin/gradle
export BREW_MAVEN=/opt/homebrew/Cellar/maven/3.9.9

# IMPORTANT: Put below environmental variable in your system or at the end of your ~/.bashrc file
#####
  export JAVA_HOME=${TARGET_DIR}
  export JDK_HOME=${JAVA_HOME}
  export GRADLE_HOME=${JAVA_HOME}/gradle
  export MAVEN_HOME=${JAVA_HOME}/maven
  export GRADLE_USER_HOME=${JAVA_HOME}/gradle.cache
  export MAVEN_USER_HOME=${JAVA_HOME}/maven.cache
  export PATH=${JAVA_HOME}/bin:${GRADLE_HOME}/bin:${MAVEN_HOME}/bin:$PATH
  export JAVA_OPTS="-Xmx1g -Xms512m"
  export GRADLE_OPTS="-Xmx2g -Dorg.gradle.daemon=true"
  export MAVEN_OPTS="-Xmx2g"
#####

# Run below only once to copy JDK-files to your target install directory
#   cp -aR ${BREW_JAVA} ${JAVA_HOME}
#   cp -aR ${BREW_GRADLE} ${GRADLE_HOME}
#   cp -aR ${BREW_MAVEN} ${MAVEN_HOME}
#   chown -R `whoami` ${TARGET_DIR}
```

After setting up the environment, verify the installation:

1. Open a terminal or command prompt and run:

```
java --version
javac --version
```


2. Expected output for JDK 21 (if you installed JDK 21 of course):

```
openjdk 21 2023-09-19
OpenJDK Runtime Environment (build 21+35-2513)
OpenJDK 64-Bit Server VM (build 21+35-2513, mixed mode, sharing)
```

3. Verify environment variables:

```
echo $JAVA_HOME  # macOS/Linux
echo %JAVA_HOME%  # Windows
```

If the output matches the expected results, the development environment is correctly configured.

1.4 More About `main` Method

The `main` method is the entry point of a Java program. It has a specific signature that the Java Virtual Machine (JVM) recognizes to start program execution.

1.4.0.0.1 Structure of the `main` Method

```
public static void main(String[] args)
```

1.4.0.0.2 Components of the `main` Method

- **`public`**: Allows the method to be accessible from anywhere.
- **`static`**: Associates the method with the class rather than an instance of the class, allowing the JVM to invoke it without creating an object.
- **`void`**: Indicates that the method does not return any value.
- **`String[] args`**: Accepts command-line arguments as an array of strings.

The `main` method acts as the starting point for executing a program. Additional methods can be invoked from within the `main` method.

The statement `System.out.println` is used to print text or data to the console.

1.4.0.0.3 Components of `System.out.println`

- **`System`**: A predefined class in the Java standard library that provides access to system-level resources.
- **`out`**: A static member of the `System` class representing the standard output stream.
- **`println`**: A method of the `PrintStream` class that prints the specified text and moves the cursor to the next line.

1.4.0.0.4 Example: Printing Text to the Console

```
System.out.println("Hello, World!");
```

This statement outputs:

```
Hello, World!
```

Please feel free to explore the code and edit the content. Let's add a few lines:

- Print multiple lines:

  ```
  System.out.println("Hello, World!");
  System.out.println("Line 1");
  System.out.println("Line 2");
  ```

  Output:

  ```
  Hello, World!
  Line 1
  Line 2
  ```

- Print text without moving to a new line:

  ```
  System.out.print("Hello");
  System.out.print(" World!");
  ```

  Output:

  ```
  Hello World!
  ```

By understanding the basic structure of a Java program, you are now ready to explore more complex Java programs and concepts.

- You can create a source file (`.java`) that defines a public class containing the `main` method.
- The `main` method serves as the program's entry point and allows the JVM to execute the program.
- The `System.out.println` statement is used to display output in the console, helping with debugging and user communication.

1.5 Understanding the Program and Debugging

Understanding the structure and behavior of a Java program is critical for effective development. This chapter focuses on key concepts such as case sensitivity, class files, debugging techniques, and command-line compilation. It also explains how to work with command-line arguments, file naming conventions, and exit codes.

Java is a case-sensitive language, meaning it distinguishes between uppercase and lowercase letters. For example:

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

In the above program:

- `System` must be capitalized; `system` will cause a compilation error.
- The class name `Example` must match the file name (`Example.java`).

When a Java program is compiled, the `javac` compiler generates a `.class` file containing the bytecode. This bytecode is executed by the Java Virtual Machine (JVM). For example:

```
javac HelloWorld.java
ls
# Output:
HelloWorld.class HelloWorld.java
```

The `HelloWorld.class` file is the compiled bytecode representation of your program.

1.5.1 Debugging with Print Statements

Debugging is the process of identifying and fixing errors in a program. A simple and effective debugging technique is using print statements:

```
public class DebugExample {
  public static void main(String[] args) {
    int x = 5;
    System.out.println("Debug: Value of x = " + x);
  }
}
```

This approach helps track variable values and program flow. However, for large applications, use a debugger provided by an IDE for more advanced features.

1.5.2 Understanding Escape Characters

Escape characters allow you to include special characters in strings. Common escape sequences include:

- `\n`: New line
- `\t`: Tab
- `\\`: Backslash
- `\"`: Double quote

```
public class EscapeCharacterExample {
  public static void main(String[] args) {
    System.out.println("Hello,\nJava!");  // New line
    System.out.println("Path: C:\\Program Files\\Java");  // Backslash
    System.out.println("She said, \"Java is awesome!\"");  // Double quote
  }
}
```

1.5.3 Using Static Methods

Static methods belong to the class rather than an instance. They can be called without creating an object:

```
public class StaticExample {
  public static void greet() {
    System.out.println("Hello from a static method!");
  }

  public static void main(String[] args) {
    greet(); // Calling the static method
  }
}
```

Static methods are often used for utility functions and entry points like `main()`.

1.5.4 Command-Line Compilation and Execution

The Java Development Kit (JDK) includes tools like `javac` (compiler) and `java` (interpreter) to compile and run Java programs directly from the command line. This process is essential for understanding Java's development workflow. Here is an example:

```
javac HelloWorld.java  # Compiles the source file
java HelloWorld        # Runs the compiled bytecode
```

Output:

```
Hello, World!
```

1.5.5 Running with Command-Line Arguments

Java programs can accept arguments from the command line, which are passed to the `main` method:

```
public class CommandLineArgs {
  public static void main(String[] args) {
    System.out.println("Argument: " + args[0]);
  }
}
```

To execute:

```
javac CommandLineArgs.java
java CommandLineArgs Hello
```

Output:

```
Argument: Hello
```

1.5.6 File Naming Rules and Case Sensitivity

The file name must match the public class name exactly, including case. For example:

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

The file must be named `HelloWorld.java`, not `helloworld.java`.

1.5.7 Exit Codes and Program Termination

Java programs can return exit codes to indicate their execution status:

```
public class ExitCodeExample {
  public static void main(String[] args) {
    System.out.println("Program completed.");
    System.exit(0); // 0 indicates success
  }
}
```

Exit codes can be used in scripts to detect success or failure:

```
java ExitCodeExample
echo $?  # Displays the exit code
```

1.6 Using IDEs for Simplified Development

As we quickly mentioned, Integrated Development Environments (IDEs) provide a powerful and user-friendly platform for writing, debugging, and managing Java programs. They simplify the development process by offering features like syntax highlighting, code suggestions, debugging tools, and integrated build systems. Popular IDEs for Java include IntelliJ IDEA, Eclipse, and Visual Studio Code. IDEs streamline the process of writing, compiling, and running Java programs:

1. Creating a Project:
   - Open your IDE (e.g., IntelliJ IDEA).
   - Create a new Java project and specify the JDK version.
   - Add a new Java class file (e.g., `HelloWorld.java`).
2. Writing Code:

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

3. Running the Program:
   - Click the **Run** button or use the IDE's shortcut (e.g., `Shift + F10` in IntelliJ).
   - The program output is displayed in the integrated console.

   Output:

```
Hello, World!
```


1.6.1 Common Compilation Errors

While writing Java programs, you might encounter compilation errors. IDEs provide features to help you identify and fix these issues efficiently:

- Syntax Errors:

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

  Error: Missing semicolon. IDEs highlight the problematic line and suggest fixes.

- Mismatched Brackets: IDEs automatically detect and warn about unmatched parentheses, braces, or brackets.
- Unresolved Symbols: If a class or method is not defined or improperly imported, the IDE provides suggestions to resolve the issue (e.g., importing the required package).
- Missing Class Name Match: File name and public class name mismatch results in errors. IDEs prompt renaming suggestions.

By leveraging real-time error detection, IDEs help reduce the time spent on debugging basic syntax issues.

1.6.2 Debugging in IDEs

Debugging is an essential skill in programming, and IDEs provide advanced debugging tools to analyze and fix issues:

1. Setting Breakpoints:
   - Click on the left margin of the code editor to set a breakpoint.
   - Execution pauses when the program reaches this line.
2. Starting the Debugger:
   - Use the **Debug** button or shortcut (e.g., `Shift + F9` in IntelliJ).
   - The IDE opens a debugging window showing the current state of variables and the call stack.

3. Step Execution:
    - Step into (`F7`): Enter the method being called.
    - Step over (`F8`): Execute the current line and move to the next.
    - Step out (`Shift + F8`): Exit the current method.
4. Inspecting Variables:
    - Hover over variables to see their current values.
    - Use the Variables tab in the debugger to inspect and modify variable values.
5. Using Watch Expressions: Add watch expressions to monitor specific variables or expressions during execution.
6. Handling Exceptions: If an exception occurs, the IDE highlights the problematic line and displays the stack trace, making it easier to identify the root cause.

Here is an example for debugging in IntelliJ IDEA: Consider the following program with a bug:

```
public class DebugExample {
  public static void main(String[] args) {
    int[] numbers = {1, 2, 3};
    System.out.println(numbers[3]); // Bug: ArrayIndexOutOfBoundsException
  }
}
```

- Set a breakpoint at `System.out.println`.
- Start the debugger. When execution pauses, inspect the `numbers` array.
- Identify that the array index is out of bounds and fix the code:

  ```
  System.out.println(numbers[2]); // Correct index
  ```

1.6.3 Advantages of Using IDEs for Debugging

- Error Highlighting: Immediate identification of syntax and semantic errors.
- Breakpoint Management: Precise control over where the program pauses.
- Call Stack Analysis: Trace the sequence of method calls leading to an issue.

- Live Variable Inspection: Observe how variable values change during execution.
- Interactive Execution: Modify variable values during debugging to test different scenarios.

1.7 Advanced Topics: Using Build Tools (Maven and Gradle)

Build tools such as **Maven** and **Gradle** are essential for managing dependencies, automating builds, and simplifying the structure of Java projects. While not mandatory for writing a basic "Hello, World!" program, understanding these tools can significantly enhance your development workflow, especially for larger and more complex projects. This section explores Maven and Gradle, their features, configurations, and advanced usage.

1.7.1 Downloading and Installing Maven and Gradle

1.7.1.0.1 Installing Maven:

1. Visit the official Apache Maven website: `https://maven.apache.org/`.
2. Download the latest binary distribution for your platform.
3. Extract the downloaded archive to a preferred directory.
4. Set the `MAVEN_HOME` environment variable:

```
export MAVEN_HOME=/path/to/maven
export PATH=$MAVEN_HOME/bin:$PATH
```

5. Verify the installation:

```
mvn --version
```

1.7.1.0.2 Installing Gradle:

1. Visit the official Gradle website: `https://gradle.org/`.
2. Download the latest binary distribution.
3. Extract the archive to a directory of your choice.

4. Set the `GRADLE_HOME` environment variable:

```
export GRADLE_HOME=/path/to/gradle
export PATH=$GRADLE_HOME/bin:$PATH
```

5. Verify the installation:

```
gradle --version
```

1.7.2 Using Maven and Gradle Build Tools

Maven is based on the Project Object Model (POM) and uses an XML configuration file (`pom.xml`) to manage dependencies, build processes, and plugins. It follows a convention-over-configuration approach, simplifying project setup.

1.7.2.0.1 Creating a Maven Project:

```
mvn archetype:generate -DgroupId=com.example -DartifactId=HelloWorld -DinteractiveMode=false
```

This generates a project structure with the necessary files and folders.

1.7.2.0.2 Example `pom.xml` File:

```
<project xmlns="http://maven.apache.org/POM/4.0.0">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.example</groupId>
        <artifactId>HelloWorld</artifactId>
        <version>1.0-SNAPSHOT</version>
        <dependencies>
                <dependency>
                        <groupId>org.junit.jupiter</groupId>
                        <artifactId>junit-jupiter</artifactId>
                        <version>5.8.2</version>
                </dependency>
        </dependencies>
</project>
```

1.7.2.0.3 Building a Maven Project:

To compile and build the project:

```
mvn clean package
```

The output JAR is located in the `target/` directory.

1.7.2.0.4 Introduction to Gradle:

Gradle uses Groovy or Kotlin scripts for configuration, making it more flexible and concise than Maven. It is widely used for modern Java projects due to its incremental build capabilities.

1.7.2.0.5 Creating a Gradle Project:

```
gradle init --type java-application
```

1.7.2.0.6 Example `build.gradle` File:

```
plugins {
        id 'java'
}

repositories {
        mavenCentral()
}

dependencies {
        implementation 'org.apache.commons:commons-lang3:3.12.0'
        testImplementation 'org.junit.jupiter:junit-jupiter:5.8.2'
}

application {
        mainClass = 'com.example.HelloWorld'
}
```

1.7.2.0.7 Building a Gradle Project:

To compile and build the project:

```
gradle build
```

The output JAR is located in the `build/libs/` directory.

1.7.3 Customizing Gradle Build Scripts

Gradle's flexibility allows developers to define custom tasks and behaviors in the build script.

1.7.3.0.1 Example: Adding a Custom Gradle Task

```
task hello {
        doLast {
                println 'Hello, Gradle!'
        }
}
```

Run the custom task:

```
gradle hello
```

Custom tasks can be used to automate repetitive operations, such as cleaning build directories or generating documentation.

1.7.4 Multi-Module Projects in Maven and Gradle

1.7.4.0.1 Multi-Module Projects in Maven: Maven supports multi-module projects where a parent project manages multiple sub-modules.

Parent `pom.xml`:

```
<modules>
<module>module1</module>
<module>module2</module>
</modules>
```

Each module has its own `pom.xml` file, and the parent project aggregates them for build and dependency management.

1.7.4.0.2 Multi-Module Projects in Gradle: Gradle handles multi-module projects using a settings file (`settings.gradle`):

```
include 'module1', 'module2'
```

Each module has its own `build.gradle` file, and the parent project orchestrates the builds.

1.7.5 Comparing Maven and Gradle

- Maven:
  - XML-based configuration (`pom.xml`).

  - Standardized, widely used in enterprise applications.
  - Limited flexibility for custom tasks.

- Gradle:
  - Groovy/Kotlin-based configuration (`build.gradle`).
  - Flexible and concise scripting for advanced customizations.
  - Faster incremental builds for large projects.

Chapter 2

Java Language: Basic Concepts

2.1 Basic Syntax and Structure of Java Programs

Understanding the basic syntax and structure of Java programs is essential for beginners to write functional and organized code. Java follows a well-defined syntax based on C-style programming, ensuring clarity, maintainability, and robustness. This chapter provides a high-level overview of the building blocks, conventions, and rules of Java programming.

2.1.0.0.1 Java Program Structure at High Level A Java program consists of classes and methods. At a minimum, a program requires a **class declaration** and a **main method**, which serves as the entry point.

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

2.1.0.0.2 Java Classes A Java program is organized into **classes**. A class serves as a blueprint for objects. It may contain methods, fields (variables), constructors, and blocks of code. The class name must match the file name for public classes.

2.1.0.0.3 The `main` Method The `main` method is the entry point for program execution. Its syntax is as follows:

```
public static void main(String[] args)
```

- `public`: Accessible from anywhere.
- `static`: Called without creating an object.
- `void`: Indicates no return value.
- `String[] args`: Accepts command-line arguments.

2.1.0.0.4 Case Sensitivity in Java Java is **case-sensitive**, meaning `System` and `system` are treated as distinct identifiers.

2.1.0.0.5 Java Statements Statements in Java are terminated with a semicolon (`;`). For example:

```
System.out.println("Hello, World!");
```


2.1.0.0.6 Comments in Java

Java supports three types of comments:

```
// Single-line comment

/* Multi-line comment */

/**
* Documentation comment for Javadoc.
*/
```

2.1.0.0.7 Identifiers and Keywords

Identifiers are names for classes, variables, and methods. They must start with a letter, underscore (_), or dollar sign ($). Keywords are reserved words like `public`, `class`, and `static`.

2.1.0.0.8 Variables in Java

Variables are used to store data. They must be declared with a data type:

```
int age = 25;
String name = "John";
```

2.1.0.0.9 Data Types

Java has two types of data:

- **Primitive Data Types**: `int`, `char`, `double`, `boolean`, etc.
- **Reference Types**: Objects, arrays, and interfaces.

2.1.0.0.10 Literals

Literals are constant values assigned to variables:

```
int a = 10;       // Integer literal
char b = 'A';     // Character literal
boolean c = true; // Boolean literal
```

2.1.0.0.11 Operators

Java provides operators for arithmetic, comparison, and logical operations:

```
int sum = 5 + 3;    // Arithmetic
boolean isEqual = (5 == 3); // Comparison
```

2.1.0.0.12 Input and Output

Output is achieved using `System.out.println`:

```
System.out.println("Output text");
```

2.1.0.0.13 Control Flow Statements

Java supports control flow structures:

- **Conditionals**: `if`, `else`, `switch`.
- **Loops**: `for`, `while`, `do-while`.

2.1.0.0.14 Example: `if-else`

```
int x = 10;
if (x > 5) {
  System.out.println("x is greater than 5");
} else {
  System.out.println("x is less than or equal to 5");
}
```

2.1.0.0.15 Example: Loops `for` loop:

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

2.1.0.0.16 Arrays Arrays store multiple values of the same type:

```
int[] numbers = {1, 2, 3, 4};
System.out.println(numbers[0]); // Output: 1
```

2.1.0.0.17 Methods in Java Methods encapsulate reusable logic:

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

2.1.0.0.18 Access Modifiers Access modifiers control visibility:

- **public**: Accessible from anywhere.
- **private**: Accessible within the class.
- **protected**: Accessible within the package and subclasses.

2.1.0.0.19 Static Members `static` variables and methods belong to the class, not instances:

```
class Example {
  static int count = 0;
  static void showCount() {
    System.out.println(count);
  }
}
```

2.1.0.0.20 Constructor Basics Constructors initialize objects:

```
class Person {
  String name;
  Person(String name) {
    this.name = name;
  }
}
```


2.1.0.0.21 Object Creation Objects are instances of classes and they are instantiated via `new` keyword.

```
Person p = new Person("John");
```

2.1.0.0.22 Packages in Java Packages group related classes:

```
package com.example;
public class MyClass { }
```

2.1.0.0.23 Importing Classes Use `import` to access classes from other packages:

```
import java.util.Scanner;
```

2.1.0.0.24 Exception Handling Java handles errors using `try-catch` blocks:

```
try {
  int result = 10 / 0;
} catch (ArithmeticException e) {
  System.out.println("Cannot divide by zero!");
}
```

2.1.0.0.25 Strings in Java Strings are immutable sequences of characters:

```
String message = "Hello";
System.out.println(message.length());
```

2.1.0.0.26 Comments and Documentation Use Javadoc comments for documentation:

```
/**
* This is a sample class.
*/
```

2.1.0.0.27 Code Blocks and Scope Curly braces define code blocks, and variables have scope limited to their block.

2.1.0.0.28 Naming Conventions Follow conventions:

- Class names: `PascalCase` (e.g., `MyClass`).
- Variables/methods: `camelCase` (e.g., `myMethod`).
- Constants: `UPPER_CASE` (e.g., `PI`).

2.1.0.0.29 Best Practices

- Write clear and readable code.
- Avoid hardcoding values.
- Use comments sparingly and meaningfully.

This program demonstrates basic syntax, method usage, and output:

```
public class Example {
  public static void main(String[] args) {
    int sum = add(5, 3);
    System.out.println("Sum: " + sum);
  }

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

2.1.1 Java Program Structure

A Java program follows a well-defined structure, which is essential for organizing code effectively, ensuring readability, and enabling smooth compilation and execution. This subsection provides a detailed explanation of Java program structure, breaking it into its key components with examples for better understanding.

2.1.1.0.1 Overview of Java Program Structure

Every Java program has the following fundamental components:

- Package declaration (optional)
- Import statements (optional)
- Class declaration (mandatory)
- `main` method (mandatory for execution)
- Methods and fields

2.1.1.0.2 Basic Java Program Template

The minimal structure of a Java program looks like this:

```
package com.example; // Optional package declaration

import java.util.Scanner; // Optional import statement

public class MyProgram { // Class declaration
  public static void main(String[] args) { // Entry point
    System.out.println("Hello, World!");
  }
}
```

2.1.1.0.3 Package Declaration

The **package declaration** defines the namespace for the class and organizes code into logical groups:

```
package com.example;

public class MyProgram {
  public static void main(String[] args) {
    System.out.println("This is in package com.example");
  }
}
```

2.1.1.0.4 Import Statements

The **import statement** allows you to use classes from other packages without fully qualifying their names:

```
import java.util.Scanner;

public class InputExample {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter a number: ");
    int num = scanner.nextInt();
    System.out.println("You entered: " + num);
  }
}
```

2.1.1.0.5 Class Declaration

A **class** is a blueprint for objects. A Java program must have at least one class:

```
public class MyClass {
  // Fields and methods go here
}
```

The class name must match the file name if it is `public`. For example, `MyClass.java` must contain `public class MyClass`.

2.1.1.0.6 Fields and Variables

Fields (class-level variables) store data:

```
public class Example {
  int number = 10; // Field

  public static void main(String[] args) {
    Example obj = new Example();
    System.out.println("Number: " + obj.number);
  }
}
```

2.1.1.0.7 Methods

Methods contain executable code. A method is defined with a return type, name, and parameters:

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

  public static void main(String[] args) {
    Example ex = new Example();
    ex.greet();
  }
}
```

2.1.1.0.8 The `main` Method

The `main` method is the program's entry point. Its signature must match:

```
public static void main(String[] args) {
  System.out.println("Program starts here!");
}
```

2.1.1.0.9 Access Modifiers

Access modifiers control visibility:

- `public`: Accessible from anywhere.
- `private`: Accessible within the class.
- `protected`: Accessible within the package or subclasses.

2.1.1.0.10 Static and Non-Static Members

Static members belong to the class, not objects:

```
public class Example {
  static int count = 0;

  public static void displayCount() {
    System.out.println("Count: " + count);
  }
```



```
  public static void main(String[] args) {
    Example.displayCount();
  }
}
```

2.1.1.0.11 Constructors

Constructors initialize objects:

```
public class Example {
  String message;

  public Example(String msg) {
    message = msg;
  }

  public static void main(String[] args) {
    Example ex = new Example("Hello!");
    System.out.println(ex.message);
  }
}
```

2.1.1.0.12 Comments in Java

Java supports single-line, multi-line, and documentation comments:

```
// Single-line comment

/* Multi-line comment */

/**
* Javadoc comment for documentation.
*/
```

2.1.1.0.13 Code Blocks and Scope

Curly braces define code blocks and variable scope:

```
public class ScopeExample {
  public static void main(String[] args) {
    int x = 5; // Local scope
    {
      int y = 10;
      System.out.println("y: " + y);
    }
    // System.out.println("y: " + y); // Error: y is out of scope
  }
}
```

2.1.1.0.14 `return` Statement

The `return` statement exits methods and returns a value:

```
public class ReturnExample {
  public static int square(int num) {
    return num * num;
```



```
  }

  public static void main(String[] args) {
    System.out.println(square(5));
  }
}
```

2.1.1.0.15 Input/Output Basics

Java uses the `Scanner` class for input and `System.out` for output:

```
import java.util.Scanner;

public class InputOutput {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter a name: ");
    String name = sc.nextLine();
    System.out.println("Hello, " + name);
  }
}
```

2.1.1.0.16 Keywords in Java

Java reserves specific words, such as `class`, `public`, `static`, `return`, etc., that cannot be used as identifiers.

2.1.1.0.17 Escape Sequences

Escape sequences handle special characters:

```
System.out.println("Hello\nWorld!");
System.out.println("Quote: \"Java\"");
```

2.1.1.0.18 Arrays in Java

Arrays store multiple values:

```
int[] numbers = {1, 2, 3};
System.out.println(numbers[0]);
```

2.1.1.0.19 Nested Classes

Classes can be nested within other classes:

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

2.1.1.0.20 Summary of Java Program Flow

- Write code with a class and `main` method.
- Compile using `javac`.
- Execute using `java`.

2.1.1.0.21 Example Program: Full Structure

```
package com.example;

import java.util.Scanner;

public class ProgramStructure {
  int number;

  public ProgramStructure(int num) {
    this.number = num;
  }

  public void displayNumber() {
    System.out.println("Number: " + number);
  }

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter a number: ");
    int input = sc.nextInt();

    ProgramStructure ps = new ProgramStructure(input);
    ps.displayNumber();
  }
}
```

2.1.2 Identifiers, Keywords, and Comments

In Java, **identifiers**, **keywords**, and **comments** play essential roles in ensuring that code is readable, structured, and executable. Identifiers are used to name program elements like classes, variables, and methods. Keywords are reserved words that have predefined meanings in the Java language, and comments allow developers to include explanatory notes in their code.

2.1.2.0.1 Identifiers in Java An **identifier** is the name used to identify classes, methods, variables, and other program elements. Examples include:

```
int age = 25;         // 'age' is an identifier
String name = "John"; // 'name' is an identifier
public class MyClass { } // 'MyClass' is an identifier
```

2.1.2.0.2 Rules for Naming Identifiers Identifiers must follow certain rules:

- Must begin with a letter (`a-z`, `A-Z`), an underscore (_), or a dollar sign ($).
- Cannot begin with a digit (0-9).
- Subsequent characters can include letters, digits, underscores, or dollar signs.

- Java keywords cannot be used as identifiers.
- Identifiers are case-sensitive.

2.1.2.0.3 Examples of Valid and Invalid Identifiers

```
// Valid identifiers
int age;
int _count;
int $price123;
String myName;

// Invalid identifiers
int 1age;        // Cannot start with a digit
int class;       // 'class' is a keyword
int my-name;     // Cannot contain hyphens
```

2.1.2.0.4 Naming Conventions for Identifiers To ensure consistency and readability, Java follows certain naming conventions:

- **Classes and Interfaces**: Use PascalCase (e.g., `MyClass`, `DataManager`).
- **Methods and Variables**: Use camelCase (e.g., `myMethod`, `studentCount`).
- **Constants**: Use UPPER_CASE with underscores (e.g., `MAX_VALUE`).

2.1.2.0.5 Examples of Naming Conventions

```
public class StudentDetails { // Class name in PascalCase
  public static final int MAX_AGE = 100; // Constant in UPPER_CASE
  private int studentId; // Variable name in camelCase

  public void displayDetails() { // Method name in camelCase
    System.out.println("Student Details");
  }
}
```

2.1.2.0.6 Java Keywords **Keywords** are reserved words with predefined meanings in Java. These cannot be used as identifiers. Examples include `class`, `public`, `static`, and `void`.

2.1.2.0.7 List of Java Keywords Java has a total of 50+ reserved keywords, some of which are:

_ (underscore)	exports	instanceof	opens	switch
abstract	extends	int	package	synchronized
assert	false	interface	permits	this
boolean	final	long	private	throw
break	finally	module	protected	throws
byte	float	native	provides	transient
case	for	new	public	true
catch	goto	non-sealed	record	try
char	if else	null	requires	uses
class	implements	opens	return	var
const	import	package	sealed	void
continue	instanceof	permits	short	volatile
default	int	private	static	while
do	interface	protected	strictfp	with
double	long	provides	super	yield

2.1.2.0.8 Using Keywords in Code

Here's an example that demonstrates some Java keywords:

```
public class KeywordExample {
  public static void main(String[] args) {
    final int MAX_COUNT = 10; // 'final' is a keyword
    for (int i = 0; i < MAX_COUNT; i++) { // 'for' keyword
      System.out.println("Count: " + i);
    }
  }
}
```

2.1.2.0.9 Reserved Literals in Java

In addition to keywords, Java has reserved literals like:

- `true`, `false`: Boolean literals.
- `null`: Represents an empty reference.

2.1.2.0.10 Comments in Java

Comments are used to add notes and explanations to the code. They are ignored during compilation and execution.

2.1.2.0.11 Single-Line Comments

Single-line comments begin with `//`:

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


2.1.2.0.12 Multi-Line Comments Multi-line comments begin with `/*` and end with `*/`:

```
/* This program demonstrates
multi-line comments in Java. */
public class CommentExample {
  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

2.1.2.0.13 Documentation Comments Documentation comments use `/** ... */` and are processed by the Javadoc tool to generate API documentation:

```
/**
* This class demonstrates Java comments.
* @author John Doe
* @version 1.0
*/
public class DocCommentExample {
  /**
  * Main method to print a message.
  * @param args Command-line arguments
  */
  public static void main(String[] args) {
    System.out.println("Documenting code with Javadoc!");
  }
}
```

2.1.2.0.14 Benefits of Comments Comments improve code readability, explain logic, and assist other developers in understanding the program.

2.1.2.0.15 Avoiding Excessive Comments While comments are useful, excessive or unnecessary comments can clutter the code.

2.1.2.0.16 Comments for Debugging Comments can temporarily disable parts of the code for debugging purposes:

```
System.out.println("Debug message");
// System.out.println("This line is commented out");
```

2.1.2.0.17 Combining Comments and Keywords Comments can describe the use of keywords:

```
public class FinalExample {
  public static final int MAX_VALUE = 100; // 'final' makes this constant
}
```


2.1.2.0.18 Commenting Complex Code Use comments to explain complex logic:

```
public class Factorial {
  public static int calculateFactorial(int n) {
    // Base case
    if (n == 0) return 1;

    // Recursive case
    return n * calculateFactorial(n - 1);
  }
}
```

2.1.2.0.19 Commenting Out Blocks of Code Block comments can disable entire sections of code:

```
/* System.out.println("Line 1");
System.out.println("Line 2"); */
```

2.1.2.0.20 Conclusion on Comments, Keywords, and Identifiers By understanding and correctly using identifiers, keywords, and comments, Java developers can write clean, readable, and maintainable programs.

2.1.3 The `main` Method and Entry Points

In Java, the `main` method serves as the entry point for program execution. It is the method that the Java Virtual Machine (JVM) looks for to start the execution of any Java application. Understanding the structure, purpose, and behavior of the `main` method is essential for writing functional Java programs.

2.1.3.0.1 What is the `main` Method? The `main` method is a predefined method in Java that acts as the entry point for program execution. It has a specific syntax that must be followed for the JVM to recognize it.

2.1.3.0.2 Basic Structure of the `main` Method The standard structure of the `main` method is:

```
public class MainExample {
  public static void main(String[] args) {
    System.out.println("Program starts here!");
  }
}
```

Here:

- **`public`**: Makes the method accessible from anywhere.
- **`static`**: Allows the JVM to call the method without creating an object of the class.
- **`void`**: Indicates that the method does not return a value.
- **`String[] args`**: An array of strings to accept command-line arguments.

2.1.3.0.3 Why is the `main` Method Static? The `main` method is declared as `static` to allow the JVM to call it without creating an instance of the class. If it were not static, an object would need to be created, which could complicate execution.

2.1.3.0.4 The Role of `String[] args` The parameter `String[] args` allows users to pass arguments to the program from the command line. These arguments are stored in an array of strings.

2.1.3.0.5 Example: Accessing Command-Line Arguments

```
public class CommandLineExample {
  public static void main(String[] args) {
    System.out.println("First argument: " + args[0]);
  }
}
```

To run this program:

```
java CommandLineExample Hello
```

Output:

```
First argument: Hello
```

2.1.3.0.6 Execution Flow of the `main` Method When you execute a Java program:

- The JVM loads the class.
- The JVM looks for the `main` method with the correct signature.
- Execution begins from the first line of the `main` method.

2.1.3.0.7 Program Without a `main` Method

A program without the `main` method will fail to execute:

```
public class NoMainExample {
  // No main method
}
```

Attempting to run this program gives:

```
Error: Main method not found
```

2.1.3.0.8 Multiple Classes with `main` Methods

Java allows multiple classes to have their own `main` methods:

```
public class FirstClass {
  public static void main(String[] args) {
    System.out.println("FirstClass main method");
  }
}

public class SecondClass {
  public static void main(String[] args) {
    System.out.println("SecondClass main method");
  }
}
```

You can execute either class:

```
java FirstClass
java SecondClass
```

2.1.3.0.9 Overloading the `main` Method

Java allows method overloading for the `main` method, but the JVM only calls the standard version:

```
public class MainOverload {
  public static void main(String[] args) {
    System.out.println("Standard main method");
  }

  public static void main(int[] numbers) {
    System.out.println("Overloaded main method");
  }
}
```

2.1.3.0.10 JVM Behavior for `main`

The JVM looks for the exact signature:

```
public static void main(String[] args)
```

If the signature is altered (e.g., missing `static` or incorrect arguments), the program will not execute.

2.1.3.0.11 Returning from `main`

You can exit the `main` method explicitly using `System.exit`:

```
public class ExitExample {
  public static void main(String[] args) {
    System.out.println("Exiting program...");
    System.exit(0);
  }
}
```

2.1.3.0.12 Accepting Multiple Command-Line Arguments

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

Run:

```
java MultiArgsExample Hello World 123
```

2.1.3.0.13 Nesting the `main` Method in Classes

The `main` method must be declared in a class. For example:

```
class Outer {
  static class Inner {
    public static void main(String[] args) {
      System.out.println("Main method in Inner class");
    }
  }
}
```

2.1.3.0.14 Using `var` in Java `main` Method (Java 10+)

Java 10 introduced `var` for type inference. However, it cannot replace the `String[] args` in the `main` method:

```
// Incorrect: var args
public static void main(var args) { }
```

2.1.3.0.15 Static Block and the `main` Method

A static block executes before the `main` method when the class is loaded:

```
public class StaticBlockExample {
  static {
    System.out.println("Static block executed");
  }
  public static void main(String[] args) {
    System.out.println("Main method executed");
  }
}
```


2.1.3.0.16 Importance of the `main` Method for Java Applications

The `main` method is critical because it allows Java to execute code independently of an IDE or environment.

2.1.3.0.17 Recursive Calls to `main`

The `main` method can call itself, leading to recursion:

```
public class RecursiveMain {
  public static void main(String[] args) {
    System.out.println("Main method");
    main(args); // Recursive call
  }
}
```

2.1.3.0.18 Handling Exceptions in the `main` Method

The `main` method can include a `try-catch` block:

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

2.1.3.0.19 Best Practices for the `main` Method

- Keep the `main` method clean and minimal.
- Use helper methods for large logic.
- Handle exceptions properly.

2.1.3.0.20 Conclusion

The `main` method is the gateway for program execution in Java. Its specific structure ensures compatibility with the JVM and makes it universally recognizable as the starting point of a program.

2.1.4 Writing Clean and Readable Code and Java Docs

Writing clean and readable code is essential for building maintainable, scalable, and collaborative software. Java provides coding conventions and tools like Javadoc to document code effectively. Following these best practices ensures that code is easy to understand, debug, and enhance over time.

2.1.4.0.1 What is Clean Code?

Clean code is code that is easy to read, understand, and modify. It follows consistent conventions, uses meaningful naming, and minimizes complexity.

2.1.4.0.2 Importance of Clean Code

Clean code offers several benefits:

- Increases readability for developers.
- Reduces bugs by simplifying logic.
- Makes maintenance easier over time.

2.1.4.0.3 Use Meaningful Names for Identifiers

Class, variable, and method names should clearly indicate their purpose:

```
// Poor naming
int x = 10;
String str = "John";

// Clean naming
int studentAge = 10;
String studentName = "John";
```

2.1.4.0.4 Naming Conventions

Java follows specific naming conventions:

- Class and Interface names: `PascalCase` (e.g., `StudentDetails`).
- Method and variable names: `camelCase` (e.g., `calculateSalary`).
- Constant names: `UPPER_CASE` (e.g., `MAX_VALUE`).

2.1.4.0.5 Avoid Magic Numbers

Replace hardcoded numbers with named constants:

```
final int MAX_STUDENTS = 50;

if (currentStudents > MAX_STUDENTS) {
  System.out.println("Class is full.");
}
```

2.1.4.0.6 Keep Methods Short and Focused

A method should perform a single responsibility:

```
// Clean method
public int add(int a, int b) {
  return a + b;
}
```


2.1.4.0.7 Avoid Long Parameter Lists

If a method requires many parameters, group them into objects:

```
class Student {
  String name;
  int age;
  String address;
}

public void registerStudent(Student student) { }
```

2.1.4.0.8 Use Proper Indentation and Formatting

Proper indentation improves readability. Most Java IDEs auto-format code using tools like:

```
Ctrl + Shift + F  # In Eclipse/IntelliJ to auto-format
```

2.1.4.0.9 Add Comments Sparingly

Comments should describe **why** something is done, not **what** is done. For example:

```
// Adding 5% bonus to the salary
double bonus = salary * 0.05;
```

2.1.4.0.10 Avoid Nested Code

Deep nesting makes code hard to follow. Use guard clauses instead:

```
// Avoid deep nesting
if (user != null) {
  if (user.isActive()) {
    processUser(user);
  }
}

// Use guard clause
if (user == null || !user.isActive()) return;
processUser(user);
```

2.1.4.0.11 Write Self-Explanatory Methods

Methods should be descriptive enough to avoid unnecessary comments:

```
public boolean isEligibleForDiscount(int age) {
  return age >= 60;
}
```

2.1.4.0.12 Use Javadoc for Code Documentation

Javadoc is a tool for generating documentation from comments written in the code. Comments begin with `/**` and end with `*/`.

2.1.4.0.13 Basic Javadoc Example

```java
/**
* Represents a student with a name and age.
*/
public class Student {
  private String name;
  private int age;

  /**
  * Creates a new student.
  * @param name The name of the student.
  * @param age The age of the student.
  */
  public Student(String name, int age) {
    this.name = name;
    this.age = age;
  }

  /**
  * Gets the student's name.
  * @return The name of the student.
  */
  public String getName() {
    return name;
  }

  /**
  * Sets the student's age.
  * @param age The new age of the student.
  */
  public void setAge(int age) {
    this.age = age;
  }
}
```

2.1.4.0.14 Tags in Javadoc Javadoc supports tags for generating structured documentation:

- `@param`: Describes method parameters.
- `@return`: Describes return values.
- `@throws`: Specifies exceptions thrown by a method.
- `@author`: Specifies the author of the class.

2.1.4.0.15 Generating Javadoc Use the `javadoc` tool to generate documentation:

```
javadoc -d docs Student.java
```


2.1.4.0.16 Avoid Redundant Comments

Avoid comments that repeat the code:

```
// Bad comment
int count = 10; // Set count to 10

// Good comment
int maxRetries = 3; // Limit retries to avoid overloading
```

2.1.4.0.17 Use Meaningful Constants for Switch Statements

Avoid using hardcoded values in switch cases:

```
final int SUCCESS = 1;
final int ERROR = 0;

switch (status) {
  case SUCCESS:
  System.out.println("Operation succeeded");
  break;
  case ERROR:
  System.out.println("Operation failed");
  break;
}
```

2.1.4.0.18 Write Testable Code

Clean code is easy to test. Use methods with clear inputs and outputs:

```
public int calculateDiscount(int price, int discountPercent) {
  return price - (price * discountPercent / 100);
}
```

2.1.4.0.19 Avoid Code Duplication

Move repeated code into reusable methods:

```
public void printGreeting(String name) {
  System.out.println("Hello, " + name);
}
```

2.1.4.0.20 Consistent Bracing Style

Use consistent bracing style to avoid confusion:

```
if (isActive) {
  System.out.println("Active");
}
```

2.1.4.0.21 Break Large Classes into Smaller Components

A class should have a single responsibility:


```
public class OrderManager {
  public void createOrder() { }
}

public class PaymentProcessor {
  public void processPayment() { }
}
```

2.1.4.0.22 Write Unit Tests with Clear Assertions

Unit tests should verify expected behavior:

```
@Test
public void testCalculateDiscount() {
  assertEquals(90, calculateDiscount(100, 10));
}
```

2.1.4.0.23 Use Final for Constants

Constants should use the `final` modifier:

```
public static final double PI = 3.14159;
```

2.1.4.0.24 Avoid Catching Generic Exceptions

Catch specific exceptions to handle errors appropriately:

```
try {
  int result = 10 / 0;
} catch (ArithmeticException e) {
  System.out.println("Cannot divide by zero.");
}
```

2.1.4.0.25 Handle Nulls Gracefully

Avoid null pointer exceptions using checks:

```
if (name != null) {
  System.out.println(name.length());
}
```

2.1.4.0.26 Use Immutable Objects Where Possible

Immutable objects simplify code and avoid unintended changes:

```
public final class ImmutableExample {
  private final String value;

  public ImmutableExample(String value) {
    this.value = value;
  }

  public String getValue() {
    return value;
  }
}
```


2.1.4.0.27 Write Clean Loops and Conditions Simplify loops and conditions:

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

2.1.4.0.28 Organize Code into Packages Use packages to organize classes logically:

```
package com.example.utils;
public class Utility { }
```

2.1.4.0.29 Follow Consistent Code Style Tools like Checkstyle and IDE formatters enforce consistent coding style.

2.1.4.0.30 Avoid Over-Engineering Keep code as simple as possible without unnecessary abstraction.

2.1.4.0.31 Use Logging Instead of `System.out` Use proper logging frameworks like SLF4J or Log4j:

```
logger.info("Application started.");
```

2.1.4.0.32 Document APIs with Javadoc Document APIs to make usage clear for other developers.

2.1.4.0.33 Avoid Excessive Comments for Simple Code If the code is self-explanatory, comments are unnecessary.

2.1.4.0.34 Conclusion Writing clean, readable code and using Javadoc ensures that Java programs are easy to maintain, understand, and collaborate on.

2.2 Data Types, Variables, and Constants

In Java, data types, variables, and constants form the foundation of how data is stored, managed, and processed. Understanding these concepts is essential for writing efficient and error-free programs. This chapter explores each of these in detail, highlighting their roles, usage, and examples.

2.2.0.0.1 What are Data Types? Data types in Java define the type of data that a variable can hold. Java is a statically typed language, meaning every variable must have a type declared at compile time.

2.2.0.0.2 Types of Data Types in Java Java data types are divided into two categories:

- **Primitive Data Types**: Built-in types that hold simple values.
- **Non-Primitive Data Types**: Types like objects, arrays, and interfaces.

2.2.0.0.3 Primitive Data Types Java has eight primitive data types:

- **byte**: 8-bit integer (`-128` to `127`).
- **short**: 16-bit integer (`-32,768` to `32,767`).
- **int**: 32-bit integer (`-2,147,483,648` to `2,147,483,647`).
- **long**: 64-bit integer.
- **float**: 32-bit floating-point number.
- **double**: 64-bit floating-point number.
- **char**: 16-bit Unicode character.
- **boolean**: Holds `true` or `false`.

2.2.0.0.4 Examples of Primitive Data Types

```
public class PrimitiveExample {
  public static void main(String[] args) {
    int age = 25;
    double price = 19.99;
    char grade = 'A';
    boolean isActive = true;

    System.out.println("Age: " + age);
    System.out.println("Price: " + price);
    System.out.println("Grade: " + grade);
    System.out.println("Active: " + isActive);
  }
}
```

2.2.0.0.5 Non-Primitive Data Types

Non-primitive data types include:

- **Strings**: Sequences of characters.
- **Arrays**: Collections of elements of the same type.
- **Classes, Interfaces, and Objects**.

2.2.0.0.6 Variables in Java

A **variable** is a container that holds data during program execution. Variables must be declared with a type and optionally initialized.

2.2.0.0.7 Declaring Variables

Variables are declared using the syntax:

```
dataType variableName = value;
```

For example:

```
int count = 10;
String name = "Alice";
```

2.2.0.0.8 Types of Variables

Java supports three types of variables:

- **Local Variables**: Declared inside methods or blocks.
- **Instance Variables**: Declared inside a class but outside methods.
- **Static Variables**: Declared with the keyword `static` and shared across all instances.

2.2.0.0.9 Example: Local and Instance Variables

```
public class VariableExample {
  int instanceVar = 100; // Instance variable

  public void show() {
    int localVar = 50; // Local variable
    System.out.println("Local: " + localVar);
    System.out.println("Instance: " + instanceVar);
  }

  public static void main(String[] args) {
    VariableExample ex = new VariableExample();
    ex.show();
  }
}
```

2.2.0.0.10 Default Values of Variables

Variables in Java have default values:

- Numeric types: `0` or `0.0`.
- `char`: `'\u0000'`.
- `boolean`: `false`.
- Objects: `null`.

2.2.0.0.11 Constants in Java

A constant is a variable whose value cannot be changed after initialization. Constants are declared using the `final` keyword:

```
final double PI = 3.14159;
System.out.println("Value of PI: " + PI);
```

2.2.0.0.12 Advantages of Using Constants

Using constants makes code more readable and prevents accidental modifications.

2.2.0.0.13 Naming Conventions for Variables and Constants

- Variables: Use camelCase (e.g., `myCount`).
- Constants: Use UPPER_CASE with underscores (e.g., `MAX_VALUE`).

2.2.0.0.14 Type Casting in Java

Type casting converts a value from one data type to another:

- **Implicit Casting**: Automatic conversion (e.g., `int` to `double`).
- **Explicit Casting**: Manual conversion using parentheses.

```
int x = 10;
double y = x;       // Implicit casting
double z = 5.5;
int a = (int) z;    // Explicit casting
```

2.2.0.0.15 Scope of Variables

The **scope** of a variable determines where it can be accessed:

- Local variables: Accessible only within the method/block.
- Instance variables: Accessible throughout the class.

2.2.0.0.16 Variable Initialization

Local variables must be initialized before use:

```
public class InitExample {
  public static void main(String[] args) {
    int value;
    // System.out.println(value); // Error: Not initialized
    value = 10;
    System.out.println(value);
  }
}
```

2.2.0.0.17 String Data Type

The `String` class represents sequences of characters:

```
String greeting = "Hello, World!";
System.out.println(greeting);
```

2.2.0.0.18 Boolean Data Type

The `boolean` type holds either `true` or `false`:

```
boolean isAvailable = true;
if (isAvailable) {
  System.out.println("It's available!");
}
```

2.2.0.0.19 Arrays as Variables

Arrays store multiple values of the same data type:

```
int[] numbers = {1, 2, 3, 4};
System.out.println(numbers[0]);
```


2.2.0.0.20 Static Variables Static variables are shared among all instances of a class:

```
public class StaticExample {
  static int count = 0;

  public StaticExample() {
    count++;
  }

  public static void main(String[] args) {
    new StaticExample();
    new StaticExample();
    System.out.println("Count: " + count); // Output: 2
  }
}
```

2.2.1 Primitive Data Types and Ranges

In Java, primitive data types are the building blocks of data manipulation. They represent simple values such as integers, floating-point numbers, characters, and boolean values. Understanding their types, sizes, and ranges is critical for writing efficient and error-free programs.

2.2.1.0.1 What are Primitive Data Types? Primitive data types in Java are predefined by the language and directly supported by the Java Virtual Machine (JVM). They are:

- **byte**
- **short**
- **int**
- **long**
- **float**
- **double**
- **char**
- **boolean**

2.2.1.0.2 Integer Data Types

Integer types store whole numbers. They include:

- **byte**: 8-bit integer.
- **short**: 16-bit integer.
- **int**: 32-bit integer.
- **long**: 64-bit integer.

2.2.1.0.3 `byte` Data Type

The `byte` type is the smallest integer type, occupying 8 bits:

- Size: 1 byte (8 bits)
- Range: `-128` to `127`

Example:

```
byte smallValue = 100;
System.out.println("Byte value: " + smallValue);
```

2.2.1.0.4 `short` Data Type

The `short` type occupies 16 bits and stores larger values than `byte`:

- Size: 2 bytes (16 bits)
- Range: `-32,768` to `32,767`

Example:

```
short mediumValue = 30000;
System.out.println("Short value: " + mediumValue);
```

2.2.1.0.5 `int` Data Type

The `int` type is the default choice for integers in Java:

- Size: 4 bytes (32 bits)
- Range: `-2,147,483,648` to `2,147,483,647`

Example:

```
int largeValue = 2000000;
System.out.println("Int value: " + largeValue);
```


2.2.1.0.6 `long` Data Type

The `long` type handles very large numbers:

- Size: 8 bytes (64 bits)
- Range: `-9,223,372,036,854,775,808` to `9,223,372,036,854,775,807`

Example:

```
long hugeValue = 10000000000L; // Add 'L' for long literals
System.out.println("Long value: " + hugeValue);
```

2.2.1.0.7 Floating-Point Data Types

Floating-point types store decimal values. Based on IEEE 754 Standard, they include:

- **float**: 32-bit floating-point number.
- **double**: 64-bit floating-point number.

2.2.1.0.8 `float` Data Type

The `float` type represents single-precision decimal values:

- Size: 4 bytes
- Range: Approximately `3.4e-038` to `3.4e+038`

Example:

```
float pi = 3.14f; // Add 'f' for float literals
System.out.println("Float value: " + pi);
```

2.2.1.0.9 `double` Data Type

The `double` type represents double-precision decimal values:

- Size: 8 bytes
- Range: Approximately `1.7e-308` to `1.7e+308`

Example:

```
double precisePi = 3.14159265359;
System.out.println("Double value: " + precisePi);
```


2.2.1.0.10 `char` Data Type

The `char` type stores a single Unicode character:

- Size: 2 bytes
- Range: `'\u0000'` to `'\uFFFF'`

Example:

```
char grade = 'A';
System.out.println("Char value: " + grade);
```

2.2.1.0.11 Unicode Representation in `char`

Characters are stored as Unicode, enabling support for multiple languages:

```
char letter = '\u0041'; // Unicode for 'A'
System.out.println("Unicode character: " + letter);
```

2.2.1.0.12 `boolean` Data Type

The `boolean` type represents truth values:

- Size: 1 bit (logical)
- Values: `true` or `false`

Example:

```
boolean isJavaFun = true;
System.out.println("Is Java fun? " + isJavaFun);
```

2.2.1.0.13 Default Values of Primitive Types

Primitive types have default values:

- **byte, short, int, long**: `0`
- **float, double**: `0.0`
- **char**: `'\u0000'`
- **boolean**: `false`

2.2.1.0.14 Type Casting

Type casting converts one data type to another:

- Implicit (widening): Smaller to larger types.
- Explicit (narrowing): Larger to smaller types.

Example:

```
int a = 10;
double b = a;     // Implicit casting
double c = 3.14;
int d = (int) c;  // Explicit casting
```


2.2.1.0.15 Memory Usage of Primitive Types Primitive types are stored in stack memory, ensuring fast access.

2.2.1.0.16 Choosing the Right Data Type Selecting the correct data type is important for:

- Optimizing memory usage.
- Ensuring accurate calculations.

2.2.1.0.17 Arithmetic Operations Primitive types support arithmetic operations:

```
int x = 10, y = 5;
System.out.println("Sum: " + (x + y));
System.out.println("Product: " + (x * y));
```

2.2.1.0.18 Primitive Type Wrapper Classes Java provides wrapper classes for primitives (e.g., `Integer`, `Double`):

```
Integer age = Integer.valueOf(25);
System.out.println("Wrapped age: " + age);
```

2.2.1.0.19 Avoiding Overflow and Underflow Performing operations on primitives may cause overflow:

```
int max = Integer.MAX_VALUE;
System.out.println("Overflow: " + (max + 1));
```

2.2.1.0.20 Primitive Types vs. Objects Primitive types are more memory-efficient compared to objects because they do not require object overhead.

2.2.2 Non-Primitive Data Types (String, Arrays)

Non-primitive data types in Java include objects, arrays, and strings. These data types are not built into the Java language directly like primitives but are derived from classes. They enable developers to store and manipulate more complex data structures. This subsection focuses on two widely used non-primitive types: Strings and Arrays, explaining their properties, usage, and examples.

2.2.2.0.1 What are Non-Primitive Data Types? Non-primitive data types are reference types that store references to memory locations where data is stored. Unlike primitives, non-primitive types can store more complex data.

2.2.2.0.2 Common Non-Primitive Data Types

- Strings: Used to represent a sequence of characters.
- Arrays: Used to store collections of elements.
- Objects: Instances of user-defined classes.

2.2.2.0.3 Introduction to Strings in Java A String in Java represents a sequence of characters. Strings are immutable, meaning their content cannot be changed after creation.

2.2.2.0.4 Declaring Strings Strings can be created in two ways:

```
String name = "Hello"; // String literal
String anotherName = new String("World"); // Using 'new' keyword
```

2.2.2.0.5 Why Strings are Immutable Once a String object is created, its content cannot be modified. This ensures thread safety and optimizes memory usage.

2.2.2.0.6 String Pool Concept String literals are stored in a special area of memory called the String pool:

```
String s1 = "Java";
String s2 = "Java"; // Points to the same memory location as s1
```

2.2.2.0.7 Common String Methods The `String` class provides methods for manipulating strings:

```
String text = "Hello, World!";
System.out.println(text.length());       // Returns the length
System.out.println(text.toUpperCase());  // Converts to upper case
System.out.println(text.contains("World")); // Checks substring
```

2.2.2.0.8 Concatenating Strings Strings can be concatenated using the `+` operator or the `concat()` method:

```
String first = "Hello";
String second = "Java";
String result = first + " " + second;
System.out.println(result); // Output: Hello Java
```


2.2.2.0.9 Comparing Strings

Strings are compared using:

- `==`: Compares references.
- `equals()`: Compares content.

```
String s1 = "Java";
String s2 = new String("Java");

System.out.println(s1 == s2);        // false (different references)
System.out.println(s1.equals(s2));   // true (same content)
```

2.2.2.0.10 StringBuilder and StringBuffer

For mutable strings, use `StringBuilder` or `StringBuffer`:

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" Java");
System.out.println(sb.toString()); // Output: Hello Java
```

2.2.2.0.11 Introduction to Arrays

An Array is a collection of elements of the same data type, stored in contiguous memory locations.

2.2.2.0.12 Declaring Arrays

Arrays are declared and initialized as follows:

```
int[] numbers = new int[5]; // Array of size 5
int[] initialized = {1, 2, 3, 4, 5}; // Predefined values
```

2.2.2.0.13 Accessing Array Elements

Array elements are accessed using indices starting from 0:

```
int[] numbers = {10, 20, 30};
System.out.println(numbers[0]); // Output: 10
```

2.2.2.0.14 Iterating Through Arrays

Use loops to iterate through an array:

```
int[] numbers = {1, 2, 3, 4};
for (int i = 0; i < numbers.length; i++) {
  System.out.println(numbers[i]);
}
```

2.2.2.0.15 Enhanced For Loop for Arrays

The enhanced `for` loop simplifies array iteration:

```
int[] numbers = {5, 10, 15};
for (int num : numbers) {
  System.out.println(num);
}
```


2.2.2.0.16 Multi-Dimensional Arrays

Arrays can have more than one dimension:

```
int[][] matrix = {
  {1, 2, 3},
  {4, 5, 6}
};
System.out.println(matrix[1][2]); // Output: 6
```

2.2.2.0.17 Default Values in Arrays

Array elements have default values:

- `int`: 0
- `float`: 0.0
- `boolean`: false
- `String`: null

2.2.2.0.18 Array Length Property

The `length` property provides the size of the array:

```
int[] numbers = new int[5];
System.out.println(numbers.length); // Output: 5
```

2.2.2.0.19 Passing Arrays to Methods

Arrays can be passed to methods as arguments:

```
public void printArray(int[] arr) {
  for (int num : arr) {
    System.out.println(num);
  }
}
```

2.2.2.0.20 Returning Arrays from Methods

Methods can return arrays:

```
public int[] createArray() {
  return new int[]{1, 2, 3};
}
```

2.2.2.0.21 Arrays vs. ArrayList

While arrays have a fixed size, `ArrayList` can dynamically grow:

```
import java.util.ArrayList;

ArrayList<Integer> list = new ArrayList<>();
list.add(10);
list.add(20);
System.out.println(list);
```


2.2.2.0.22 Sorting Arrays

Arrays can be sorted using `Arrays.sort()`:

```
import java.util.Arrays;

int[] numbers = {3, 1, 4, 2};
Arrays.sort(numbers);
System.out.println(Arrays.toString(numbers));
```

2.2.2.0.23 Searching Arrays

Use `Arrays.binarySearch()` for efficient searching:

```
int[] numbers = {1, 2, 3, 4};
int index = Arrays.binarySearch(numbers, 3);
System.out.println("Index: " + index);
```

2.2.2.0.24 Cloning Arrays

Arrays can be cloned to create copies:

```
int[] original = {1, 2, 3};
int[] copy = original.clone();
System.out.println(Arrays.toString(copy));
```

2.2.2.0.25 Arrays of Objects

Arrays can hold object references:

```
String[] names = {"Alice", "Bob"};
System.out.println(names[0]);
```

2.2.2.0.26 Null Values in Arrays

Elements in arrays of objects default to `null`:

```
String[] array = new String[2];
System.out.println(array[0]); // Output: null
```

2.2.2.0.27 Ragged Arrays (Uneven Rows)

Multi-dimensional arrays can have uneven rows:

```
int[][] ragged = {
  {1, 2},
  {3, 4, 5}
};
```

2.2.2.0.28 Comparing Arrays

Use `Arrays.equals()` to compare arrays:

```
int[] a = {1, 2};
int[] b = {1, 2};
System.out.println(Arrays.equals(a, b)); // true
```

2.2.2.0.29 Conclusion: Strings vs. Arrays

Strings are immutable sequences of characters, while arrays are fixed-size collections of elements. Both are critical for handling and storing data effectively.

2.2.3 Variable Scope and Lifetime

In Java, variables have a **scope** and **lifetime** that determine where the variable can be accessed and how long it exists in memory. Understanding these concepts is essential for writing clean and efficient programs. This subsection explains the different scopes, lifetimes, and best practices for variable usage in Java.

2.2.3.0.1 What is Variable Scope? Variable scope defines the portion of a program where a variable can be accessed. Variables can have different scopes depending on where and how they are declared.

2.2.3.0.2 Types of Variable Scope in Java Java supports four types of variable scope:

- **Local Scope**: Variables declared inside methods or blocks.
- **Instance Scope**: Variables declared at the class level, outside methods.
- **Static/Class Scope**: Variables declared as `static` within a class.
- **Block Scope**: Variables declared within a specific block (e.g., loops or conditionals).

2.2.3.0.3 Local Variables and Their Scope Local variables are declared within a method, constructor, or block and are accessible only within that specific block. They are created when the method/block starts and destroyed when it ends.

```
public class LocalScopeExample {
  public void display() {
    int localVar = 10; // Local variable
    System.out.println("Local Variable: " + localVar);
  }

  public static void main(String[] args) {
    LocalScopeExample obj = new LocalScopeExample();
    obj.display();
    // System.out.println(localVar); // Error: localVar not accessible here
  }
}
```

2.2.3.0.4 Lifetime of Local Variables Local variables exist only during the execution of their enclosing method or block. Once the method finishes, the variable is removed from memory.

2.2.3.0.5 Instance Variables and Their Scope Instance variables are declared at the class level but outside any method. They are tied to an object and can be accessed using the object reference.

```
public class InstanceScopeExample {
  int instanceVar = 5; // Instance variable

  public void display() {
    System.out.println("Instance Variable: " + instanceVar);
  }

  public static void main(String[] args) {
    InstanceScopeExample obj = new InstanceScopeExample();
    obj.display(); // Accessible through the object
  }
}
```

2.2.3.0.6 Lifetime of Instance Variables Instance variables are created when an object is instantiated and exist until the object is destroyed or becomes eligible for garbage collection.

2.2.3.0.7 Static Variables and Their Scope Static variables are declared with the `static` keyword. They belong to the class, not to any specific object, and can be accessed directly using the class name.

```
public class StaticScopeExample {
  static int staticVar = 100; // Static variable

  public static void main(String[] args) {
    System.out.println("Static Variable: " + StaticScopeExample.staticVar);
  }
}
```

2.2.3.0.8 Lifetime of Static Variables Static variables are created when the class is loaded into memory and exist until the program terminates.

2.2.3.0.9 Block Scope Variables declared within blocks (e.g., loops or conditionals) are limited to that block. They are destroyed when the block ends.

```
public class BlockScopeExample {
  public static void main(String[] args) {
    if (true) {
      int blockVar = 20; // Block scope
      System.out.println("Block Variable: " + blockVar);
    }
    // System.out.println(blockVar); // Error: blockVar not accessible here
  }
}
```


2.2.3.0.10 Shadowing Variables

A local variable can **shadow** an instance variable if both have the same name within the same scope. Use the `this` keyword to refer to the instance variable.

```
public class ShadowingExample {
  int number = 10; // Instance variable

  public void display() {
    int number = 20; // Local variable shadows instance variable
    System.out.println("Local: " + number);
    System.out.println("Instance: " + this.number);
  }

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

2.2.3.0.11 Best Practices for Variable Scope

- Minimize the scope of variables to the smallest possible block.
- Use instance variables only when they are required across methods.
- Avoid global static variables unless absolutely necessary.

2.2.3.0.12 Scope in Loops and Conditionals

Variables declared inside a loop are destroyed after the loop ends:

```
for (int i = 0; i < 5; i++) {
  System.out.println("Loop Variable: " + i);
}
// System.out.println(i); // Error: i is not accessible here
```

2.2.3.0.13 Final Variables

A variable can be declared `final`, making it a constant that cannot be reassigned:

```
final int MAX_VALUE = 100;
System.out.println("Final Variable: " + MAX_VALUE);
```

2.2.3.0.14 Garbage Collection and Variable Lifetime

Instance variables are destroyed when the object they belong to becomes unreachable. The garbage collector frees up the memory.

2.2.3.0.15 Static Blocks and Static Variable Initialization Static variables can be initialized in static blocks:

```
public class StaticBlockExample {
  static int value;

  static {
    value = 50; // Static block
    System.out.println("Static block initialized");
  }

  public static void main(String[] args) {
    System.out.println("Static Value: " + value);
  }
}
```

2.2.3.0.16 Summary of Variable Types and Scope

- Local Variables: Limited to a method/block and short-lived.
- Instance Variables: Tied to objects and persist as long as the object exists.
- Static Variables: Shared across all objects and exist for the program's duration.
- Block Variables: Limited to specific blocks (e.g., loops and conditionals).

2.2.3.0.17 Conclusion Understanding variable scope and lifetime ensures efficient memory usage and cleaner code. By minimizing scope and managing variables effectively, you avoid unintended behavior and improve program performance.

2.2.4 Type Casting and Type Conversion

Type casting and type conversion in Java allow developers to convert a variable of one data type into another. These conversions are necessary when dealing with incompatible types or optimizing memory usage. Understanding the nuances of type casting and conversion ensures proper handling of data without introducing errors.

2.2.4.0.1 What is Type Conversion? Type conversion refers to converting a variable from one data type to another. In Java, type conversion can happen automatically (implicit conversion) or manually (explicit casting).

2.2.4.0.2 Types of Type Conversion

Type conversion in Java is categorized into two types:

- **Implicit Conversion (Widening)**: Automatic conversion of a smaller type to a larger type.
- **Explicit Casting (Narrowing)**: Manual conversion of a larger type to a smaller type.

2.2.4.0.3 Implicit Type Conversion (Widening)

Widening happens automatically when there is no risk of data loss. For example:

```
public class ImplicitConversion {
  public static void main(String[] args) {
    int intValue = 100;
    double doubleValue = intValue; // Automatic conversion
    System.out.println("Integer: " + intValue);
    System.out.println("Double: " + doubleValue);
  }
}
```

2.2.4.0.4 Rules for Implicit Conversion

Widening occurs in the following order of types:

$$\texttt{byte} \rightarrow \texttt{short} \rightarrow \texttt{int} \rightarrow \texttt{long} \rightarrow \texttt{float} \rightarrow \texttt{double}$$

2.2.4.0.5 Example of Widening Conversion

```
byte b = 10;
int i = b; // Widening byte to int
System.out.println("Widened Value: " + i);
```

2.2.4.0.6 Explicit Type Casting (Narrowing)

Narrowing requires explicit casting because it may result in data loss:

```
public class ExplicitCasting {
  public static void main(String[] args) {
    double doubleValue = 9.78;
    int intValue = (int) doubleValue; // Manual casting
    System.out.println("Double: " + doubleValue);
    System.out.println("Integer: " + intValue);
  }
}
```

2.2.4.0.7 Rules for Explicit Casting

Narrowing follows the reverse order of widening and must be done manually:

$$\texttt{double} \rightarrow \texttt{float} \rightarrow \texttt{long} \rightarrow \texttt{int} \rightarrow \texttt{short} \rightarrow \texttt{byte}$$

2.2.4.0.8 Data Loss in Explicit Casting

When narrowing, precision might be lost:

```
double value = 123.456;
int converted = (int) value; // Decimal part is truncated
System.out.println("Converted Value: " + converted);
```

2.2.4.0.9 Casting Between Integer and Character

A `char` can be cast to an `int` and vice versa:

```
char letter = 'A';
int ascii = letter; // Implicit conversion
System.out.println("ASCII value of A: " + ascii);

int number = 66;
char letterB = (char) number; // Explicit conversion
System.out.println("Character: " + letterB);
```

2.2.4.0.10 Casting Between `int` and `boolean`

Java does not allow casting between `int` and `boolean`. The following is invalid:

```
// int i = 1;
// boolean b = (boolean) i; // Compilation error
```

2.2.4.0.11 Type Conversion with Strings

A String can be converted to numeric types using parsing methods:

```
public class StringConversion {
  public static void main(String[] args) {
    String number = "123";
    int value = Integer.parseInt(number);
    System.out.println("Converted Integer: " + value);
  }
}
```

2.2.4.0.12 Converting Numeric Types to Strings

Numeric values can be converted to Strings using `String.valueOf()`:

```
int num = 100;
String text = String.valueOf(num);
System.out.println("String value: " + text);
```

2.2.4.0.13 Automatic Type Promotion in Expressions

During arithmetic operations, smaller types are promoted to larger types automatically:

```
byte a = 10;
byte b = 20;
int result = a + b; // Both promoted to int
System.out.println("Result: " + result);
```


2.2.4.0.14 Type Conversion in Method Overloading

Method overloading resolves methods based on the type conversion:

```
public class MethodOverload {
  void display(int x) {
    System.out.println("Integer: " + x);
  }
  void display(double x) {
    System.out.println("Double: " + x);
  }

  public static void main(String[] args) {
    MethodOverload obj = new MethodOverload();
    obj.display(10);    // Calls integer version
    obj.display(10.5);  // Calls double version
  }
}
```

2.2.4.0.15 Wrapper Classes for Conversion

Java provides wrapper classes like `Integer`, `Double`, and `Character` to convert between primitive types and objects:

```
int number = 100;
Integer wrapped = Integer.valueOf(number); // Boxing
int unwrapped = wrapped.intValue();        // Unboxing
System.out.println("Boxed: " + wrapped + ", Unboxed: " + unwrapped);
```

2.2.4.0.16 Conversion Between `float` and `int`

Explicit casting is required when converting from `float` to `int`:

```
float value = 12.34f;
int result = (int) value;
System.out.println("Float to Int: " + result);
```

2.2.4.0.17 Casting and Arrays

Type casting can also occur in arrays, but compatibility must be ensured:

```
double[] doubleArray = {1.1, 2.2, 3.3};
int[] intArray = new int[doubleArray.length];
for (int i = 0; i < doubleArray.length; i++) {
  intArray[i] = (int) doubleArray[i];
}
```

2.2.4.0.18 Using `instanceof` for Safe Casting

The `instanceof` operator checks type compatibility before casting:

```
Object obj = "Hello";
if (obj instanceof String) {
  String text = (String) obj;
  System.out.println("Casted String: " + text);
}
```


2.2.4.0.19 Runtime Errors with Casting

Invalid casting leads to a `ClassCastException`:

```
Object obj = new Integer(10);
String text = (String) obj; // Causes runtime exception
```

2.2.4.0.20 Best Practices for Type Conversion and Casting

- Use widening (implicit) conversions where possible.
- Avoid narrowing unless necessary and check for precision loss.
- Use `instanceof` before casting objects.

2.2.4.0.21 Avoiding Data Loss in Casting

When narrowing, ensure that the value fits within the target type:

```
int bigValue = 300;
byte smallValue = (byte) bigValue; // Overflow: unexpected result
System.out.println("Result: " + smallValue);
```

2.2.4.0.22 Using Generics to Avoid Casting

Generics eliminate the need for explicit casting:

```
ArrayList<String> list = new ArrayList<>();
list.add("Hello");
String text = list.get(0); // No casting required
```

2.2.4.0.23 Type Promotion in Mixed Data Types

When performing operations on mixed data types, smaller types are promoted:

```
int a = 5;
float b = 2.5f;
float result = a + b; // 'a' is promoted to float
System.out.println("Result: " + result);
```

2.2.4.0.24 Safe Type Conversion with BigDecimal

For precise numeric conversions, use `BigDecimal`:

```
BigDecimal value = new BigDecimal("123.456");
System.out.println(value.intValue());
```

2.2.4.0.25 Summary of Casting

Type conversion and type casting are essential for working with incompatible data types in Java. Developers must handle explicit casting carefully to avoid runtime errors and precision loss.

2.2.4.0.26 Conclusion

By understanding implicit and explicit type conversions, developers can ensure smooth and error-free data manipulation, enhancing the accuracy and reliability of Java programs.

2.3 Operators and Expressions

In Java, operators are special symbols or keywords that perform operations on variables and values to produce a result. Expressions are combinations of variables, operators, and literals that evaluate to a single value. Understanding operators and expressions is fundamental to writing logic in Java programs.

2.3.0.0.1 What are Operators? An operator performs a specific operation on one or more operands (values or variables). Java provides various types of operators to manipulate data and perform computations.

2.3.0.0.2 Types of Operators in Java Java operators are categorized as follows:

- Arithmetic Operators
- Relational (Comparison) Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Unary Operators
- Ternary Operator

2.3.0.0.3 Arithmetic Operators Arithmetic operators perform basic mathematical operations:

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


```
public class ArithmeticExample {
  public static void main(String[] args) {
    int a = 10, b = 3;
    System.out.println("Addition: " + (a + b));
    System.out.println("Subtraction: " + (a - b));
    System.out.println("Multiplication: " + (a * b));
    System.out.println("Division: " + (a / b));
    System.out.println("Modulus: " + (a % b));
  }
}
```

2.3.0.0.4 Relational Operators

Relational operators compare two values and return a boolean result:

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

```
int x = 10, y = 5;
System.out.println(x > y);  // true
System.out.println(x == y); // false
```

2.3.0.0.5 Logical Operators

Logical operators are used for boolean logic:

- `&&`: Logical AND
- `||`: Logical OR
- `!`: Logical NOT

```
boolean a = true, b = false;
System.out.println(a && b); // false
System.out.println(a || b); // true
System.out.println(!a);     // false
```


2.3.0.0.6 Bitwise Operators

Bitwise operators perform operations on bits:

- `&`: Bitwise AND
- `|`: Bitwise OR
- `^`: Bitwise XOR
- `~`: Bitwise Complement
- `<<`: Left shift
- `>>`: Right shift

2.3.0.0.7 Assignment Operators

Assignment operators assign values to variables:

- `=`: Simple assignment
- `+=`, `-=`, `*=`, `/=`: Compound assignments

```
int a = 5;
a += 3; // Equivalent to a = a + 3
System.out.println(a); // Output: 8
```

2.3.0.0.8 Unary Operators

Unary operators act on a single operand:

- `+`: Unary plus
- `-`: Unary minus
- `++`: Increment
- `--`: Decrement
- `!`: Logical NOT

```
int x = 5;
System.out.println(++x); // Pre-increment: 6
System.out.println(x--); // Post-decrement: 6, then 5
```

2.3.0.0.9 Ternary Operator

The ternary operator (`? :`) is a shorthand for `if-else` statements:

```
int a = 10, b = 20;
int max = (a > b) ? a : b; // Returns the larger value
System.out.println("Max: " + max);
```


2.3.0.0.10 Operator Precedence and Associativity Operator precedence determines the order in which operators are executed. Associativity defines the direction of execution:

- Precedence: Multiplication `*`, Division `/`, and Modulus `%` come before Addition `+` and Subtraction `-`.
- Associativity: Left-to-right for most operators.

```
int result = 10 + 5 * 2; // Multiplication happens first
System.out.println("Result: " + result); // Output: 20
```

2.3.0.0.11 Expressions in Java An expression is a combination of variables, literals, and operators that evaluates to a single value:

```
int x = 10;
int y = 5;
int result = x * y + 10; // Expression
System.out.println("Result: " + result);
```

2.3.0.0.12 Types of Expressions Expressions in Java include:

- Arithmetic expressions: Use arithmetic operators.
- Relational expressions: Compare values using relational operators.
- Logical expressions: Use logical operators to evaluate conditions.

2.3.0.0.13 Evaluating Expressions Java evaluates expressions from left to right, following operator precedence:

```
int a = 10, b = 20, c = 5;
int result = a + b * c;
System.out.println("Result: " + result); // Output: 110
```

2.3.0.0.14 Compound Expressions Expressions can combine multiple operations:

```
int a = 5, b = 10, c = 2;
int result = (a + b) / c;
System.out.println("Result: " + result); // Output: 7
```

2.3.0.0.15 Casting in Expressions Explicit casting can be used to control the result of an expression:

```
int a = 10, b = 3;
double result = (double) a / b;
System.out.println("Result: " + result); // Output: 3.333
```


2.3.0.0.16 String Concatenation with +

The + operator can also concatenate strings:

```
String name = "John";
int age = 25;
System.out.println("Name: " + name + ", Age: " + age);
```

2.3.0.0.17 Short-Circuit Operators

The logical operators && and || support short forms:

```
int x = 5;
if (x > 3 || ++x > 10) {
  System.out.println("Condition is true");
}
System.out.println(x); // Output: 5, because '||' short-circuits
```

2.3.0.0.18 Best Practices for Using Operators

- Use parentheses to clarify precedence.
- Avoid deep nesting of expressions.
- Avoid mixing incompatible types without proper casting.

2.3.1 Arithmetic, Relational, and Logical Operators

2.3.1.0.1 Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations on numerical values. These include:

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

```
public class ArithmeticExample {
  public static void main(String[] args) {
    int a = 15, b = 4;
    System.out.println("Addition: " + (a + b));       // Output: 19
    System.out.println("Subtraction: " + (a - b));    // Output: 11
    System.out.println("Multiplication: " + (a * b)); // Output: 60
    System.out.println("Division: " + (a / b));       // Output: 3
    System.out.println("Modulus: " + (a % b));        // Output: 3
  }
}
```


2.3.1.0.2 Division and Modulus Behavior

The `/` operator performs integer division when both operands are integers:

```
int result = 10 / 3; // Result: 3 (fractional part truncated)
```

The `%` operator returns the remainder:

```
System.out.println(10 % 3); // Output: 1
```

2.3.1.0.3 Relational Operators

Relational operators compare two values and return a boolean result:

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

```
int x = 10, y = 20;
System.out.println(x == y); // false
System.out.println(x != y); // true
System.out.println(x < y);  // true
System.out.println(x >= y); // false
```

2.3.1.0.4 Logical Operators

Logical operators are used to evaluate boolean expressions:

- `&&`: Logical AND
- `||`: Logical OR
- `!`: Logical NOT

```
boolean a = true, b = false;
System.out.println(a && b); // false
System.out.println(a || b); // true
System.out.println(!a);     // false
```


2.3.2 Increment/Decrement and Assignment Operators

2.3.2.0.1 Increment and Decrement Operators The increment (`++`) and decrement (`--`) operators increase or decrease the value of a variable by 1:

- **Pre-increment (`++x`)**: Increments before using the value.
- **Post-increment (`x++`)**: Uses the value, then increments.
- **Pre-decrement (`--x`)**: Decrements before using the value.
- **Post-decrement (`x--`)**: Uses the value, then decrements.

```
int x = 5;
System.out.println(++x); // Pre-increment: Output 6
System.out.println(x--); // Post-decrement: Output 6, x becomes 5
```

2.3.2.0.2 Assignment Operators Assignment operators assign values to variables. They include:

- **`=`**: Simple assignment
- **`+=`**, **`-=`**, **`*=`**, **`/=`**, **`%=`**: Compound assignments

```
int a = 10;
a += 5; // a = a + 5
System.out.println(a); // Output: 15
```

2.3.3 Conditional (Ternary) Operators

2.3.3.0.1 Ternary Operator Syntax The ternary operator (`? :`) is a shorthand for `if-else` statements:

```
condition ? expression1 : expression2;
```

2.3.3.0.2 Example of Ternary Operator

```
int a = 10, b = 20;
int max = (a > b) ? a : b; // Checks which value is larger
System.out.println("Max: " + max); // Output: 20
```

2.3.3.0.3 Nested Ternary Operator Ternary operators can be nested for multiple conditions:

```
int marks = 85;
String grade = (marks >= 90) ? "A" :
(marks >= 75) ? "B" : "C";
System.out.println("Grade: " + grade); // Output: B
```


2.3.4 Operator Precedence and Associativity

2.3.4.0.1 Operator Precedence Operator precedence determines the order in which operators are evaluated. Operators with higher precedence are executed first.

- Multiplication `*`, Division `/`, Modulus `%` have higher precedence than Addition `+` and Subtraction `-`.
- Logical AND `&&` has higher precedence than Logical OR `||`.

```
int result = 10 + 5 * 2; // Multiplication happens first
System.out.println("Result: " + result); // Output: 20
```

2.3.4.0.2 Associativity of Operators When operators have the same precedence, associativity determines the evaluation order:

- **Left-to-Right**: Most operators, like arithmetic and logical operators.
- **Right-to-Left**: Unary operators (`++`, `--`) and assignment operators (`=`, `+=`).

2.3.4.0.3 Example of Left-to-Right Associativity

```
int result = 100 / 5 * 2;
System.out.println("Result: " + result); // Left-to-right: Output 40
```

2.3.4.0.4 Example of Right-to-Left Associativity

```
int x = 5;
x += x -= x * 2; // Evaluates right to left
System.out.println("Result: " + x); // Output: -5
```

2.3.4.0.5 Parentheses to Control Precedence Use parentheses to explicitly specify the order of operations:

```
int result = (10 + 5) * 2;
System.out.println("Result: " + result); // Output: 30
```

2.3.4.0.6 Mixing Relational and Logical Operators Relational operators are evaluated before logical operators:

```
int a = 5, b = 10, c = 15;
boolean result = a < b && b < c;
System.out.println(result); // Output: true
```


2.3.4.0.7 Best Practices for Operators

- Use parentheses to avoid confusion in complex expressions.
- Avoid deeply nested ternary operators.
- Ensure you understand operator precedence to prevent logical errors.

2.3.4.0.8 Conclusion By understanding arithmetic, relational, logical, increment/decrement, assignment, and ternary operators, developers can write clear and concise code. Proper use of operator precedence and associativity ensures that expressions evaluate as expected.

2.4 Control Flow: Conditionals

Control flow statements in Java allow developers to dictate the order in which code is executed based on conditions. Conditionals such as `if`, `else`, and `switch` enable branching in programs, allowing logic to adapt dynamically to different inputs.

2.4.1 Conditionals: `if`, `else`, and `else if`

2.4.1.0.1 The `if` Statement The `if` statement checks a condition and executes a block of code if the condition is `true`.

```
public class IfExample {
  public static void main(String[] args) {
    int number = 10;

    if (number > 0) {
      System.out.println("The number is positive.");
    }
  }
}
```

2.4.1.0.2 The `else` Statement The `else` block is executed when the `if` condition is `false`.

```
public class ElseExample {
  public static void main(String[] args) {
    int number = -5;

    if (number > 0) {
      System.out.println("Positive number");
    } else {
      System.out.println("Negative number");
    }
  }
}
```

2.4.1.0.3 The `else if` Ladder The `else if` ladder allows checking multiple conditions sequentially.

```
public class ElseIfExample {
  public static void main(String[] args) {
    int score = 75;

    if (score >= 90) {
      System.out.println("Grade: A");
    } else if (score >= 75) {
      System.out.println("Grade: B");
    } else {
      System.out.println("Grade: C");
    }
  }
}
```


2.4.2 Simple and Nested `if` Statements

2.4.2.0.1 Simple `if` Statement A simple `if` checks one condition.

```
if (x > 0) {
  System.out.println("x is positive.");
}
```

2.4.2.0.2 Nested `if` Statements A nested `if` occurs when one `if` statement is inside another. This allows checking conditions in layers.

```
public class NestedIfExample {
  public static void main(String[] args) {
    int number = 10;

    if (number > 0) {
      if (number % 2 == 0) {
        System.out.println("Positive even number");
      } else {
        System.out.println("Positive odd number");
      }
    } else {
      System.out.println("Number is non-positive");
    }
  }
}
```

2.4.3 Switch Statements and Pattern Matching

2.4.3.0.1 The `switch` Statement The `switch` statement executes one block of code based on the value of an expression.

```
public class SwitchExample {
  public static void main(String[] args) {
    int day = 3;

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


2.4.3.0.2 Enhanced `switch` (Java 12+)

The enhanced `switch` allows cleaner syntax using arrow notation.

```
public class EnhancedSwitch {
  public static void main(String[] args) {
    int day = 2;

    switch (day) {
      case 1 -> System.out.println("Monday");
      case 2 -> System.out.println("Tuesday");
      case 3 -> System.out.println("Wednesday");
      default -> System.out.println("Invalid day");
    }
  }
}
```

2.4.3.0.3 Pattern Matching in `switch` (Java 17+)

Pattern matching simplifies type checks within `switch` statements.

```
public class PatternMatchingSwitch {
  public static void main(String[] args) {
    Object obj = "Hello";

    switch (obj) {
      case String s -> System.out.println("It's a string: " + s);
      case Integer i -> System.out.println("It's an integer: " + i);
      default -> System.out.println("Unknown type");
    }
  }
}
```

2.4.4 Best Practices for Conditional Logic

2.4.4.0.1 Avoid Deep Nesting

Deeply nested `if` statements make code harder to read. Use guard clauses or `return` statements instead.

```
// Avoid deep nesting
if (x != null) {
  if (x.isActive()) {
    System.out.println("Processing...");
  }
}

// Use guard clauses
if (x == null || !x.isActive()) return;
System.out.println("Processing...");
```

2.4.4.0.2 Use `switch` for Multiple Conditions

Prefer `switch` statements when dealing with multiple discrete values for clarity and performance.

2.4.4.0.3 Use the Ternary Operator for Simplicity The ternary operator is ideal for simple conditions that return a value.

```
int x = 10;
String result = (x > 0) ? "Positive" : "Non-positive";
System.out.println(result);
```

2.4.4.0.4 Combine Conditions Thoughtfully Avoid redundant conditions by combining them with logical operators.

```
if (age >= 18 && age <= 60) {
  System.out.println("Eligible");
}
```

2.4.4.0.5 Be Cautious with Equality Comparisons For object comparisons, use `equals()` instead of `==`.

```
String a = "test";
if (a.equals("test")) {
  System.out.println("Strings match");
}
```

2.4.4.0.6 Avoid Fall-Through in `switch` Always use `break` to prevent fall-through behavior in `switch`.

```
switch (option) {
  case 1:
  System.out.println("Option 1");
  break;
  case 2:
  System.out.println("Option 2");
  break;
  default:
  System.out.println("Invalid");
}
```

2.4.4.0.7 Use `default` in `switch` Always include a `default` case to handle unexpected values.

2.4.4.0.8 Optimize Conditions for Readability Write conditions in an order that improves readability and logic flow.

2.4.4.0.9 Leverage Modern Features Use modern features like enhanced `switch` and pattern matching to write cleaner, more maintainable code.

2.5 Loops in Java

Loops in Java are used to execute a block of code repeatedly as long as a condition is true. Java provides several types of loops, including **`for`**, **`while`**, and **`do-while`**, each suitable for different scenarios. This chapter explores these loops in detail, along with advanced concepts like `break`, `continue`, labeled loops, and infinite loops.

2.5.1 Basic Loop Constructs

2.5.1.0.1 The `for` Loop The `for` loop is used when the number of iterations is known beforehand. It has three parts: initialization, condition, and update.

```
public class ForLoopExample {
  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) { // Initialization, condition, update
      System.out.println("Iteration: " + i);
    }
  }
}
```

2.5.1.0.2 The `while` Loop The `while` loop executes as long as the given condition is true. It is useful when the number of iterations is unknown.

```
public class WhileLoopExample {
  public static void main(String[] args) {
    int i = 1;
    while (i <= 5) { // Condition check
      System.out.println("Iteration: " + i);
      i++; // Update
    }
  }
}
```

2.5.1.0.3 The `do-while` Loop The `do-while` loop guarantees at least one execution because the condition is checked after the loop body.

```
public class DoWhileExample {
  public static void main(String[] args) {
    int i = 1;
    do {
      System.out.println("Iteration: " + i);
      i++;
    } while (i <= 5); // Condition check at the end
  }
}
```


2.5.2 Enhanced `for` Loop (For-Each)

2.5.2.0.1 What is the Enhanced `for` Loop? The enhanced `for` loop, also called the `for-each` loop, is used to iterate over collections or arrays without managing an index.

2.5.2.0.2 Example: Iterating Through an Array

```
public class ForEachExample {
  public static void main(String[] args) {
    int[] numbers = {1, 2, 3, 4, 5};

    for (int num : numbers) { // For-each loop
      System.out.println("Number: " + num);
    }
  }
}
```

2.5.2.0.3 Limitations of For-Each Loop The `for-each` loop cannot:

- Modify the index of the loop.
- Traverse in reverse order.

2.5.3 `break`, `continue`, and Labeled Loops

2.5.3.0.1 The `break` Statement The `break` statement terminates the loop immediately.

```
public class BreakExample {
  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      if (i == 3) break; // Exit loop when i equals 3
      System.out.println("Iteration: " + i);
    }
  }
}
```

2.5.3.0.2 The `continue` Statement The `continue` statement skips the current iteration and continues with the next iteration.

```
public class ContinueExample {
  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {
      if (i == 3) continue; // Skip iteration when i equals 3
      System.out.println("Iteration: " + i);
    }
  }
}
```


2.5.3.0.3 Labeled Loops

Labeled loops allow breaking or continuing outer loops from within nested loops.

```
public class LabeledLoopExample {
  public static void main(String[] args) {
    outer: for (int i = 1; i <= 3; i++) {
      for (int j = 1; j <= 3; j++) {
        if (j == 2) continue outer; // Skip to next iteration of outer loop
        System.out.println("i = " + i + ", j = " + j);
      }
    }
  }
}
```

2.5.4 Infinite Loops

2.5.4.0.1 Infinite Loops Using `for`

A loop becomes infinite when the condition always evaluates to `true`.

```
public class InfiniteForLoop {
  public static void main(String[] args) {
    for (;;) { // No condition specified
      System.out.println("This is an infinite loop.");
      break; // Use break to terminate
    }
  }
}
```

2.5.4.0.2 Infinite Loops Using `while`

```
public class InfiniteWhileLoop {
  public static void main(String[] args) {
    while (true) { // Condition always true
      System.out.println("This is an infinite loop.");
      break; // Use break to exit
    }
  }
}
```

2.5.4.0.3 Infinite Loops: Risks and Uses

Infinite loops are useful in event-driven systems but can cause a program to hang if no exit condition is defined.

2.5.5 Best Practices for Loops

2.5.5.0.1 Use `for` for Known Iterations

When the number of iterations is fixed, prefer the `for` loop for clarity.

2.5.5.0.2 Use `while` for Unknown Iterations If the condition depends on dynamic input, the `while` loop is a better choice.

2.5.5.0.3 Avoid Hardcoding Conditions in Loops Avoid magic numbers; use constants or variables for conditions.

```
final int MAX = 10;
for (int i = 0; i < MAX; i++) {
  System.out.println(i);
}
```

2.5.5.0.4 Minimize Deep Nesting in Loops Excessive nesting makes loops difficult to read. Use helper methods or labeled loops where appropriate.

2.5.5.0.5 Break Infinite Loops Safely Always include exit conditions or `break` statements to prevent accidental infinite loops.

2.5.5.0.6 Use Enhanced `for` Loop for Collections The `for-each` loop simplifies iteration when index manipulation is unnecessary.

2.5.5.0.7 Handle Edge Cases in Loops Ensure loops handle edge cases, such as empty arrays or zero iterations, without errors.

2.5.5.0.8 Avoid Unnecessary Code Inside Loops Keep loops efficient by minimizing operations performed in each iteration.

```
for (int i = 0; i < 1000; i++) {
  System.out.println("Processing " + i); // Avoid unnecessary calculations
}
```

2.5.5.0.9 Use Labeled Loops Sparingly Labeled loops can improve clarity for nested iterations but should be used only when necessary to avoid confusion.

2.5.6 Loop Optimization

Loop optimization focuses on improving the efficiency of loops to minimize execution time and resource usage. Inefficient loops can significantly impact performance, especially when dealing with large datasets or time-sensitive applications. This subsection covers techniques and best practices for optimizing loops in Java.

2.5.6.0.1 Minimize Repeated Calculations Avoid performing redundant operations or calculations within the loop body. Move invariant expressions outside the loop to save processing time.

```
// Inefficient loop
for (int i = 0; i < list.size(); i++) {
  System.out.println(list.get(i));
}

// Optimized loop
int size = list.size();
for (int i = 0; i < size; i++) {
  System.out.println(list.get(i));
}
```

2.5.6.0.2 Use Enhanced `for` Loop for Collections and Arrays The enhanced `for` loop (for-each) is more efficient and concise when iterating over arrays or collections. It avoids manual index management.

```
// Optimized iteration using enhanced for loop
for (String name : names) {
  System.out.println(name);
}
```

2.5.6.0.3 Prefer `ArrayList` over Linked Data Structures for Indexed Access For indexed iteration, `ArrayList` provides faster access compared to linked data structures like `LinkedList`.

```
// Efficient indexed access
ArrayList<Integer> numbers = new ArrayList<>();
for (int i = 0; i < numbers.size(); i++) {
  System.out.println(numbers.get(i));
}
```

2.5.6.0.4 Use `break` and `continue` Wisely Use `break` to exit a loop early and `continue` to skip unnecessary iterations. This avoids redundant checks.

```
for (int i = 0; i < 10; i++) {
  if (i == 5) break; // Exit when i is 5
  if (i % 2 == 0) continue; // Skip even numbers
  System.out.println(i);
}
```

2.5.6.0.5 Avoid String Concatenation Inside Loops String concatenation in loops creates new String objects in memory, causing performance issues. Use `StringBuilder` instead.


```java
// Inefficient String concatenation
String result = "";
for (int i = 0; i < 5; i++) {
  result += i;
}

// Optimized with StringBuilder
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 5; i++) {
  sb.append(i);
}
String result = sb.toString();
```

2.5.6.0.6 Reduce Loop Condition Overhead

The loop condition is evaluated at each iteration. Simplify it where possible, or precompute its value.

```java
// Inefficient condition
for (int i = 0; i < array.length; i++) {
  System.out.println(array[i]);
}

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

2.5.6.0.7 Use Local Variables for Repeated Access

Cache frequently accessed data (e.g., object fields) in local variables for better performance.

```java
// Accessing fields repeatedly
for (int i = 0; i < items.length; i++) {
  System.out.println(items[i].value);
}

// Cache in a local variable
for (int i = 0; i < items.length; i++) {
  Item item = items[i];
  System.out.println(item.value);
}
```

2.5.6.0.8 Opt for `for` Loops Over `while` When Appropriate

`for` loops make initialization, condition checking, and updates explicit, reducing the chance of errors.

```java
for (int i = 0; i < 10; i++) {
  System.out.println(i);
}
```


2.5.6.0.9 Avoid Unnecessary Object Creation

Creating objects inside a loop can degrade performance. Reuse objects where possible.

```
// Inefficient object creation
for (int i = 0; i < 10; i++) {
  Person p = new Person("Name");
}

// Optimized object reuse
Person p = new Person("Name");
for (int i = 0; i < 10; i++) {
  p.setId(i);
}
```

2.5.6.0.10 Use Parallel Streams for Large Datasets

For large collections, use Java's `parallelStream` to leverage multi-core processing.

```
list.parallelStream().forEach(System.out::println);
```

2.5.6.0.11 Combine Conditions to Reduce Checks

Combine multiple conditions using logical operators to avoid redundant evaluations.

```
for (int i = 0; i < 100; i++) {
  if (i % 2 == 0 && i % 3 == 0) {
    System.out.println(i);
  }
}
```

2.5.6.0.12 Remove Unnecessary Loop Operations

Avoid complex operations inside the loop that could be precomputed outside.

```
// Inefficient
for (int i = 0; i < 100; i++) {
  System.out.println(Math.pow(i, 2));
}

// Precompute outside the loop if needed
double[] squares = new double[100];
for (int i = 0; i < 100; i++) {
  squares[i] = Math.pow(i, 2);
}
```

2.5.6.0.13 Use Labeled Loops for Early Exit in Nested Loops

Labeled loops allow you to break or continue outer loops without additional conditions.

```
outer: for (int i = 0; i < 5; i++) {
  for (int j = 0; j < 5; j++) {
    if (j == 2) break outer;
    System.out.println(i + ", " + j);
  }
}
```


2.5.6.0.14 Avoid Infinite Loops Unless Required While infinite loops can be useful, ensure they have clear exit conditions to prevent program hangs.

```
while (true) {
  if (someCondition) break; // Safe exit condition
}
```

2.5.6.0.15 Use Compiler Optimizations The Java compiler optimizes loops during compilation. Use simple constructs and avoid overly complex conditions to help the compiler.

2.5.6.0.16 Avoid Excessive Recursion For tasks that can be solved iteratively, prefer loops over recursion to avoid stack overflow errors and memory overhead.

2.5.6.0.17 Test Loop Performance for Large Data Always test loops with realistic input sizes to identify bottlenecks. Use profiling tools like VisualVM or JMH for performance measurement. By following these best practices, developers can write efficient, clean, and performant loops. Optimizing loop conditions, caching values, and avoiding unnecessary operations significantly reduces execution time, especially in performance-critical applications.

2.6 Introduction to Methods

Methods in Java are blocks of code designed to perform specific tasks. They promote code reusability, modularity, and organization, allowing developers to break down large programs into manageable parts. This chapter provides a detailed explanation of methods, including their definition, invocation, argument passing, return types, and best practices.

2.6.1 Defining and Calling Methods

2.6.1.0.1 What is a Method? A method is a named block of code that can be executed when called. Methods perform a specific operation and help avoid code duplication.

2.6.1.0.2 Syntax for Defining a Method The general syntax of a method is:

```
returnType methodName(parameters) {
  // Method body
  // Code to perform the task
  return value; // Optional, based on return type
}
```

- **returnType**: The data type of the value the method returns (`void` if nothing is returned).
- **methodName**: The name of the method.
- **parameters**: Input values passed to the method.
- **return**: Optional statement that specifies the value to return.

2.6.1.0.3 Example of Defining and Calling a Method

```
public class MethodExample {
  // Method definition
  public static int add(int a, int b) {
    return a + b;
  }

  public static void main(String[] args) {
    // Method call
    int result = add(5, 3);
    System.out.println("Sum: " + result);
  }
}
```


2.6.2 Passing Arguments: By Value vs By Reference

2.6.2.0.1 Argument Passing in Java Java uses **pass-by-value** for method arguments. This means that the method receives a copy of the value, and changes made inside the method do not affect the original variable.

2.6.2.0.2 Passing Primitive Types (By Value) When primitives are passed, their values are copied:

```
public class PassByValue {
  public static void modifyValue(int x) {
    x = x * 2; // Modify local copy
  }

  public static void main(String[] args) {
    int num = 10;
    modifyValue(num);
    System.out.println("Original Value: " + num); // Output: 10
  }
}
```

2.6.2.0.3 Passing Object References When objects are passed, the reference (memory address) is copied. Changes to object properties affect the original object:

```
class Person {
  String name;
}

public class PassByReference {
  public static void changeName(Person p) {
    p.name = "Alice";
  }

  public static void main(String[] args) {
    Person person = new Person();
    person.name = "John";
    changeName(person);
    System.out.println("Name: " + person.name); // Output: Alice
  }
}
```

2.6.3 Return Types and Void Methods

2.6.3.0.1 Return Types in Methods A method can return a value of any type, including primitives, objects, or no value (`void`).

2.6.3.0.2 Methods with a Return Type Methods that return a value must include a `return` statement:


```
public static int square(int x) {
  return x * x; // Return the square of x
}

public static void main(String[] args) {
  int result = square(5);
  System.out.println("Square: " + result);
}
```

2.6.3.0.3 Void Methods (No Return Value)

A `void` method does not return any value and does not include a `return` statement:

```
public static void greet(String name) {
  System.out.println("Hello, " + name);
}

public static void main(String[] args) {
  greet("John");
}
```

2.6.4 Method Overloading

2.6.4.0.1 What is Method Overloading?

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

2.6.4.0.2 Example of Method Overloading

```
public class OverloadingExample {
  // Method 1
  public static int add(int a, int b) {
    return a + b;
  }

  // Method 2
  public static double add(double a, double b) {
    return a + b;
  }

  public static void main(String[] args) {
    System.out.println("Sum (int): " + add(5, 3));
    System.out.println("Sum (double): " + add(2.5, 3.5));
  }
}
```

2.6.5 Method Design and Best Practices

2.6.5.0.1 Keep Methods Small and Focused

A method should perform a single, well-defined task. Avoid combining unrelated logic.


```
// Good practice
public static double calculateCircleArea(double radius) {
  return Math.PI * radius * radius;
}
```

2.6.5.0.2 Use Descriptive Method Names Choose meaningful method names that reflect what the method does.

```
// Descriptive method name
public static boolean isEven(int number) {
  return number % 2 == 0;
}
```

2.6.5.0.3 Minimize the Number of Parameters Limit parameters to 3-4 where possible. For more parameters, consider using objects to group related data.

```
// Too many parameters
public static void registerUser(String name, String email, int age, String address) { }

// Better approach
class User {
  String name, email, address;
  int age;
}
public static void registerUser(User user) { }
```

2.6.5.0.4 Avoid Code Duplication Reuse methods instead of duplicating code. It improves maintainability.

```
public static int square(int x) {
  return x * x;
}

public static int cube(int x) {
  return x * square(x);
}
```

2.6.5.0.5 Use Return Statements Appropriately Avoid multiple `return` statements unless they improve readability.

```
// Good practice
public static boolean isPositive(int number) {
  return number > 0;
}
```


2.6.5.0.6 Document Methods Using Javadoc Use Javadoc comments to describe the purpose, parameters, and return values of methods.

```
/
* Calculates the sum of two integers.
* @param a First integer
* @param b Second integer
* @return The sum of a and b
*/
public static int add(int a, int b) {
  return a + b;
}
```

2.6.5.0.7 Test Methods Thoroughly Write unit tests to ensure methods behave as expected with various inputs.

```
@Test
public void testAdd() {
  assertEquals(8, add(5, 3));
}
```

Also make sure to write effective methods:

- Keep methods small and single-purpose.
- Use meaningful method names.
- Limit the number of parameters.
- Reuse code and avoid duplication.
- Document methods using Javadoc.

2.7 Arrays and Multi-Dimensional Arrays

Arrays in Java are used to store multiple values of the same data type in a single variable. They are a fundamental data structure that provides a way to organize and manipulate data efficiently. Java also supports multi-dimensional arrays for more complex data representation, such as matrices.

2.7.1 Declaring and Initializing Arrays

2.7.1.0.1 What is an Array? An array is a fixed-size collection of elements of the same data type. Arrays are stored in contiguous memory locations, enabling fast access.

2.7.1.0.2 Declaring an Array Arrays are declared using the following syntax:

```
dataType[] arrayName;
```

2.7.1.0.3 Initializing an Array Arrays can be initialized in two ways:

- Explicit Initialization: Assign values directly during declaration.
- Dynamic Initialization: Define the size and add elements later.

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

// Dynamic Initialization
int[] values = new int[5];
values[0] = 10;
values[1] = 20;
System.out.println(values[0]); // Output: 10
```

2.7.1.0.4 Default Values in Arrays If no values are assigned, array elements have default values:

- `int`, `float`, `double`: 0
- `char`: `'\u0000'`
- `boolean`: `false`
- Objects: `null`

2.7.2 Accessing and Iterating Through Arrays

2.7.2.0.1 Accessing Array Elements Array elements are accessed using indices, starting from 0:

```
int[] numbers = {5, 10, 15, 20};
System.out.println(numbers[0]); // Output: 5
System.out.println(numbers[3]); // Output: 20
```

2.7.2.0.2 Iterating Through Arrays with Loops Use `for` loops or enhanced `for` loops to iterate through arrays:

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

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

// Enhanced for loop (for-each)
for (int num : numbers) {
  System.out.println(num);
}
```

2.7.3 Multi-Dimensional Arrays and Matrices

2.7.3.0.1 What are Multi-Dimensional Arrays? A multi-dimensional array is an array of arrays. The most common type is a two-dimensional array, used to represent matrices.

2.7.3.0.2 Declaring and Initializing a 2D Array Two-dimensional arrays can be declared and initialized as follows:

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

// Dynamic Initialization
int[][] grid = new int[2][3];
grid[0][0] = 1;
grid[1][2] = 5;
```

2.7.3.0.3 Accessing Elements in a 2D Array Use two indices to access elements:

```
System.out.println(matrix[1][2]); // Output: 6
```


2.7.3.0.4 Iterating Through a 2D Array

Use nested loops to traverse a two-dimensional array:

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

for (int i = 0; i < matrix.length; i++) { // Rows
  for (int j = 0; j < matrix[i].length; j++) { // Columns
    System.out.print(matrix[i][j] + " ");
  }
  System.out.println();
}
```

2.7.3.0.5 Jagged Arrays

Jagged arrays are arrays with rows of different lengths:

```
int[][] jaggedArray = {
  {1, 2},
  {3, 4, 5},
  {6}
};

for (int[] row : jaggedArray) {
  for (int num : row) {
    System.out.print(num + " ");
  }
  System.out.println();
}
```

2.7.4 Array Operations and Utility Methods

2.7.4.0.1 Sorting Arrays

The `Arrays.sort()` method sorts arrays in ascending order:

```
import java.util.Arrays;

int[] numbers = {5, 3, 8, 1, 2};
Arrays.sort(numbers);
System.out.println(Arrays.toString(numbers)); // Output: [1, 2, 3, 5, 8]
```

2.7.4.0.2 Searching Arrays

Use `Arrays.binarySearch()` to search for an element in a sorted array:

```
int[] numbers = {1, 2, 3, 4, 5};
int index = Arrays.binarySearch(numbers, 4);
System.out.println("Index: " + index); // Output: 3
```


2.7.4.0.3 Copying Arrays

The `Arrays.copyOf()` method copies an array:

```
int[] original = {1, 2, 3};
int[] copy = Arrays.copyOf(original, original.length);
System.out.println(Arrays.toString(copy)); // Output: [1, 2, 3]
```

2.7.4.0.4 Filling Arrays

The `Arrays.fill()` method fills all elements of an array with a specific value:

```
int[] array = new int[5];
Arrays.fill(array, 7);
System.out.println(Arrays.toString(array)); // Output: [7, 7, 7, 7, 7]
```

2.7.4.0.5 Comparing Arrays

The `Arrays.equals()` method checks if two arrays are equal:

```
int[] a = {1, 2, 3};
int[] b = {1, 2, 3};
System.out.println(Arrays.equals(a, b)); // Output: true
```

2.7.4.0.6 Converting Arrays to Strings

The `Arrays.toString()` method converts an array to a string:

```
int[] array = {1, 2, 3};
System.out.println(Arrays.toString(array)); // Output: [1, 2, 3]
```

2.7.4.0.7 Multi-Dimensional Arrays and `Arrays.deepToString()`

For multi-dimensional arrays, use `Arrays.deepToString()`:

```
int[][] matrix = {{1, 2}, {3, 4}};
System.out.println(Arrays.deepToString(matrix)); // Output: [[1, 2], [3, 4]]
```

2.7.4.0.8 Best Practices for Using Arrays

- Use meaningful names for arrays to describe their purpose.
- Avoid hardcoding array sizes; use constants or dynamic values.
- Use enhanced `for` loops for cleaner iteration when index access is unnecessary.
- Use utility methods from `java.util.Arrays` for common operations like sorting, copying, and searching.

Chapter 3

Java Language: Intermediate Concepts

3.1 Method Overloading and Recursion

Methods in Java are powerful tools for modularizing code and improving reusability. This chapter covers two advanced method concepts: Method Overloading and Recursion. Method overloading allows multiple methods with the same name but different parameter lists, while recursion enables methods to call themselves to solve problems.

3.1.1 Understanding Method Overloading

3.1.1.0.1 What is Method Overloading? Method overloading occurs when two or more methods in the same class have the same name but different parameter lists. The compiler determines which method to call based on the number and types of arguments passed.

3.1.1.0.2 Why Use Method Overloading? Method overloading improves code readability and usability by allowing a method to handle different types or numbers of inputs without renaming the method.

3.1.1.0.3 Example of Method Overloading In this example, the `add()` method is overloaded to accept different parameter types:

```
public class OverloadingExample {
  // Method 1: Adds two integers
  public int add(int a, int b) {
    return a + b;
  }

  // Method 2: Adds two doubles
  public double add(double a, double b) {
    return a + b;
  }

  // Method 3: Adds three integers
  public int add(int a, int b, int c) {
    return a + b + c;
  }

  public static void main(String[] args) {
    OverloadingExample obj = new OverloadingExample();

    System.out.println("Sum (int): " + obj.add(5, 10));
    System.out.println("Sum (double): " + obj.add(5.5, 2.3));
    System.out.println("Sum (three ints): " + obj.add(1, 2, 3));
  }
}
```


3.1.1.0.4 Rules for Method Overloading

- Methods must have the same name.
- Methods must differ in the number or types of parameters.
- Return type alone cannot distinguish overloaded methods.

3.1.1.0.5 Common Use Cases for Method Overloading

- Handling different data types with the same logic.
- Providing multiple options for input parameters.
- Simplifying method naming for similar operations.

3.1.2 Recursion and Base Cases

3.1.2.0.1 What is Recursion? Recursion is a programming technique where a method calls itself to solve a smaller instance of the same problem.

3.1.2.0.2 Structure of a Recursive Method A recursive method consists of:

- A **base case** that terminates the recursion.
- A **recursive step** that reduces the problem size and calls the method again.

3.1.2.0.3 Example: Factorial Using Recursion The factorial of a number `n` is defined as `n! = n × (n-1)!`.

```
public class FactorialExample {
  public static int factorial(int n) {
    if (n == 0) { // Base case
      return 1;
    } else { // Recursive step
      return n * factorial(n - 1);
    }
  }

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


3.1.2.0.4 Importance of Base Cases The base case prevents infinite recursion by defining when the recursion should stop. Omitting a base case will lead to a `StackOverflowError`.

3.1.3 Tail Recursion and Optimized Recursion

3.1.3.0.1 What is Tail Recursion? Tail recursion occurs when the recursive call is the last statement in the method. Tail-recursive methods are more efficient because they do not require additional stack frames.

3.1.3.0.2 Example: Tail Recursion for Factorial

```
public class TailRecursionExample {
  public static int factorial(int n, int result) {
    if (n == 0) { // Base case
      return result;
    }
    return factorial(n - 1, n * result); // Tail-recursive call
  }

  public static void main(String[] args) {
    int result = factorial(5, 1);
    System.out.println("Factorial of 5: " + result);
  }
}
```

Here, the result is passed as an argument, reducing the need for multiple stack frames.

3.1.3.0.3 Benefits of Tail Recursion

- Optimizes memory usage by reusing stack frames.
- Improves performance for deep recursion.

3.1.3.0.4 Tail Recursion vs Regular Recursion In regular recursion, additional operations occur after the recursive call, leading to extra stack frames.

3.1.4 Comparing Recursion and Iteration

3.1.4.0.1 Recursion vs Iteration: Key Differences

- Recursion solves problems by repeatedly calling the method itself.
- Iteration uses loops (`for`, `while`) to repeat a block of code.

3.1.4.0.2 Example: Factorial Using Iteration

The factorial problem can also be solved using a loop:

```
public class IterativeFactorial {
  public static int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
      result *= i;
    }
    return result;
  }

  public static void main(String[] args) {
    System.out.println("Factorial of 5: " + factorial(5));
  }
}
```

3.1.4.0.3 Advantages of Recursion

- Simplifies complex problems like tree traversal and backtracking.
- Code is cleaner and easier to understand for certain problems.

3.1.4.0.4 Disadvantages of Recursion

- Recursion consumes more memory due to stack frames.
- Deep recursion can cause stack overflow errors.
- Iterative solutions are often more efficient for simpler problems.

3.1.4.0.5 When to Use Recursion vs Iteration

- Use recursion for problems involving hierarchical or divide-and-conquer structures (e.g., trees, graphs).
- Use iteration for problems with predictable, linear repetition (e.g., loops).

3.1.5 Best Practices for Recursion and Overloading

3.1.5.0.1 Best Practices for Method Overloading

- Ensure overloaded methods differ in parameter types or counts.
- Avoid excessive overloading, which can confuse code readers.
- Use clear and descriptive method names.

3.1.5.0.2 Best Practices for Recursion

- Always define a base case to avoid infinite recursion.
- Use tail recursion where possible for optimization.
- Analyze recursion depth and memory usage for large inputs.

3.1.5.0.3 Summary of Method Overloading and Recursion Method overloading enhances code readability and usability, while recursion simplifies complex problems. Choosing between recursion and iteration depends on the problem structure and performance requirements.

3.2 Introduction to Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that models real-world entities using objects. It focuses on organizing code into modular units, making programs easier to design, maintain, and scale. Java is a fully object-oriented language that leverages OOP principles to achieve clean and efficient programming.

3.2.1 Core OOP Principles

OOP is based on four main principles:

- **Encapsulation**: Bundling data (fields) and methods into a single unit (class) and restricting access to certain parts of an object.
- **Inheritance**: Allowing one class to inherit properties and behaviors (methods) from another class.
- **Polymorphism**: Providing a single interface to represent different forms of behavior.
- **Abstraction**: Hiding implementation details and exposing only the essential functionalities.

3.2.2 Classes vs Objects

3.2.2.0.1 What is a Class? A **class** is a blueprint or template that defines the properties (fields) and behaviors (methods) of objects. Classes do not consume memory until an object is created.

3.2.2.0.2 What is an Object? An **object** is an instance of a class. It represents a real-world entity with state and behavior. Objects consume memory and store actual data.

3.2.2.0.3 Example: Class and Object

```
class Car { // Class
  String brand;
  int speed;

  // Method to display car details
```



```
  void displayDetails() {
    System.out.println("Brand: " + brand + ", Speed: " + speed);
  }
}

public class Main {
  public static void main(String[] args) {
    Car car1 = new Car(); // Object creation
    car1.brand = "Toyota";
    car1.speed = 120;

    Car car2 = new Car();
    car2.brand = "Honda";
    car2.speed = 100;

    car1.displayDetails();
    car2.displayDetails();
  }
}
```

Output:

```
Brand: Toyota, Speed: 120
Brand: Honda, Speed: 100
```

3.2.3 Encapsulation

3.2.3.0.1 What is Encapsulation? Encapsulation refers to the practice of bundling data (fields) and methods within a class and controlling access to them using access modifiers like `private`, `public`, `protected`.

3.2.3.0.2 Example of Encapsulation

```
class Person {
  // Private fields
  private String name;
  private int age;

  // Public getters and setters
  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public int getAge() {
    return age;
  }

  public void setAge(int age) {
```



```
    this.age = age;
  }
}

public class Main {
  public static void main(String[] args) {
    Person person = new Person();
    person.setName("Alice");
    person.setAge(25);

    System.out.println("Name: " + person.getName());
    System.out.println("Age: " + person.getAge());
  }
}
```

Encapsulation ensures that fields cannot be accessed directly, improving data security and integrity.

3.2.4 Inheritance

3.2.4.0.1 What is Inheritance? Inheritance allows one class (child) to inherit the properties and methods of another class (parent). It promotes code reuse.

3.2.4.0.2 Example of Inheritance

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

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

public class Main {
  public static void main(String[] args) {
    Dog dog = new Dog();
    dog.eat();  // Inherited method
    dog.bark(); // Child-specific method
  }
}
```

Output:

```
Eating...
Barking...
```


3.2.5 Polymorphism

3.2.5.0.1 What is Polymorphism? Polymorphism allows a single method, class, or interface to represent multiple forms. It can be achieved through:

- Method Overloading: Same method name with different parameters.
- Method Overriding: Subclass provides its implementation for a method defined in the parent class.

3.2.5.0.2 Example: Method Overloading

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

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

public class Main {
  public static void main(String[] args) {
    Calculator calc = new Calculator();
    System.out.println(calc.add(2, 3));       // Calls int version
    System.out.println(calc.add(2.5, 3.5));   // Calls double version
  }
}
```

3.2.5.0.3 Example: Method Overriding

```
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 Main {
  public static void main(String[] args) {
    Animal animal = new Dog();
    animal.sound(); // Calls the overridden method in Dog
  }
}
```


3.2.6 Abstraction

3.2.6.0.1 What is Abstraction? Abstraction hides the implementation details of a class and only exposes its essential functionality. It can be achieved using:

- Abstract classes (`abstract` keyword)
- Interfaces

3.2.6.0.2 Example: Abstract Class

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

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

public class Main {
  public static void main(String[] args) {
    Shape shape = new Circle();
    shape.draw();
  }
}
```

3.2.6.0.3 Example: Interface

```
interface Animal {
  void sound();
}

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

public class Main {
  public static void main(String[] args) {
    Animal cat = new Cat();
    cat.sound();
  }
}
```

3.2.7 Designing Real-World Classes

3.2.7.0.1 Identifying Classes and Objects In a real-world scenario, identify entities as classes and their properties/behaviors as fields and methods.

3.2.7.0.2 Example: Designing a Bank Account Class

```
class BankAccount {
  private String accountNumber;
  private double balance;

  public BankAccount(String accountNumber) {
    this.accountNumber = accountNumber;
    this.balance = 0.0;
  }

  public void deposit(double amount) {
    balance += amount;
  }

  public void withdraw(double amount) {
    if (amount <= balance) {
      balance -= amount;
    } else {
      System.out.println("Insufficient funds");
    }
  }

  public void displayBalance() {
    System.out.println("Account Number: " + accountNumber);
    System.out.println("Balance: $" + balance);
  }
}

public class Main {
  public static void main(String[] args) {
    BankAccount account = new BankAccount("12345");
    account.deposit(500);
    account.withdraw(200);
    account.displayBalance();
  }
}
```

3.2.7.0.3 Best Practices for OOP Design

- Use encapsulation to protect fields and provide controlled access.
- Favor composition over inheritance for better flexibility.
- Design classes with single responsibilities.
- Use descriptive and meaningful names for classes and methods.

3.2.7.0.4 Summary of OOP Principles OOP in Java revolves around encapsulation, inheritance, polymorphism, and abstraction. These principles simplify code organization, improve reusability, and enhance maintainability. Understanding and applying these concepts is key to writing clean and scalable programs.

3.3 Classes, Objects, and Constructors

In Java, classes, objects, and constructors are the foundational building blocks of object-oriented programming. This chapter explores how to define classes, create objects, use constructors, and understand the lifecycle of objects, including garbage collection.

3.3.1 Defining Classes and Creating Objects

3.3.1.0.1 What is a Class? A class is a blueprint or template for creating objects. It defines the properties (fields) and behaviors (methods) that objects of the class will have.

3.3.1.0.2 Syntax for Defining a Class

```
class ClassName {
  // Fields (properties)
  dataType fieldName;

  // Methods (behaviors)
  returnType methodName(parameters) {
    // Method body
  }
}
```

3.3.1.0.3 What is an Object? An object is an instance of a class that contains actual data and allows access to the methods defined in the class.

3.3.1.0.4 Creating Objects from a Class To create an object, use the `new` keyword followed by a call to the class constructor.

```
class Car {
  // Fields
  String brand;
  int speed;

  // Method
  void displayDetails() {
    System.out.println("Brand: " + brand + ", Speed: " + speed);
  }
}

public class Main {
  public static void main(String[] args) {
    // Creating objects
    Car car1 = new Car();
    Car car2 = new Car();
```



```
    // Assigning values to fields
    car1.brand = "Toyota";
    car1.speed = 120;

    car2.brand = "Honda";
    car2.speed = 100;

    // Calling methods
    car1.displayDetails();
    car2.displayDetails();
  }
}
```

Output:

```
Brand: Toyota, Speed: 120
Brand: Honda, Speed: 100
```

3.3.2 Constructors: Default, Parameterized, and Copy

3.3.2.0.1 What is a Constructor? A constructor is a special method that is invoked automatically when an object is created. It initializes the object's fields.

3.3.2.0.2 Features of Constructors

- Constructors have the same name as the class.
- They do not have a return type (not even `void`).
- A class can have multiple constructors (constructor overloading).

3.3.2.0.3 Default Constructor If no constructor is defined, Java provides a default constructor that initializes fields with default values (e.g., 0 for numbers, `null` for objects).

```
class Person {
  String name;
  int age;

  // Default constructor
  Person() {
    System.out.println("Default constructor called");
  }

  void display() {
    System.out.println("Name: " + name + ", Age: " + age);
  }
```



```
}

public class Main {
  public static void main(String[] args) {
    Person p = new Person(); // Default constructor invoked
    p.display();
  }
}
```

3.3.2.0.4 Parameterized Constructor

A parameterized constructor initializes fields with user-supplied values.

```
class Person {
  String name;
  int age;

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

  void display() {
    System.out.println("Name: " + name + ", Age: " + age);
  }
}

public class Main {
  public static void main(String[] args) {
    Person p1 = new Person("Alice", 25);
    Person p2 = new Person("Bob", 30);

    p1.display();
    p2.display();
  }
}
```

3.3.2.0.5 Copy Constructor

A copy constructor creates a new object by copying the values of an existing object.

```
class Person {
  String name;
  int age;

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

  // Copy constructor
  Person(Person p) {
    this.name = p.name;
    this.age = p.age;
```



```
  }

  void display() {
    System.out.println("Name: " + name + ", Age: " + age);
  }
}

public class Main {
  public static void main(String[] args) {
    Person p1 = new Person("Alice", 25); // Original object
    Person p2 = new Person(p1);          // Copy constructor

    p1.display();
    p2.display();
  }
}
```

Output:

```
Name: Alice, Age: 25
Name: Alice, Age: 25
```

3.3.3 Overloading Constructors

3.3.3.0.1 What is Constructor Overloading? Constructor overloading occurs when a class has multiple constructors with different parameter lists. This allows the creation of objects in multiple ways.

3.3.3.0.2 Example of Constructor Overloading

```
class Person {
  String name;
  int age;

  // Default constructor
  Person() {
    this.name = "Unknown";
    this.age = 0;
  }

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

  void display() {
    System.out.println("Name: " + name + ", Age: " + age);
  }
}

public class Main {
```



```
  public static void main(String[] args) {
    Person p1 = new Person();                // Default constructor
    Person p2 = new Person("Alice", 25);     // Parameterized constructor

    p1.display();
    p2.display();
  }
}
```

Output:

```
Name: Unknown, Age: 0
Name: Alice, Age: 25
```

3.3.4 Garbage Collection and Object Lifecycle

3.3.4.0.1 Object Lifecycle in Java The lifecycle of an object includes:

1. Creation: Using the `new` keyword.
2. Usage: Performing operations through methods and fields.
3. Destruction: When the object is no longer needed, it becomes eligible for garbage collection.

3.3.4.0.2 What is Garbage Collection? Garbage collection (GC) is an automatic process in Java that reclaims memory occupied by unused objects. The JVM determines when an object is unreachable and removes it to free up memory.

3.3.4.0.3 Example of Garbage Collection The `finalize()` method is called just before the object is garbage-collected:

```
class Person {
  String name;

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

  @Override
  protected void finalize() throws Throwable {
    System.out.println("Object " + name + " is being garbage collected");
  }
}

public class Main {
  public static void main(String[] args) {
    Person p1 = new Person("Alice");
```



```
        Person p2 = new Person("Bob");

        // Dereference objects
        p1 = null;
        p2 = null;

        // Request garbage collection
        System.gc();

        System.out.println("End of program");
    }
}
```

Output:

```
Object Alice is being garbage collected
Object Bob is being garbage collected
End of program
```

3.3.4.0.4 Best Practices for Constructors and Objects

- Always initialize object fields using constructors.
- Avoid writing unnecessary `finalize()` methods as GC is automatic.
- Use constructor overloading to provide flexibility for object creation.
- Avoid creating objects unnecessarily to reduce memory usage.

3.3.4.0.5 Summary Classes define the blueprint for objects, while constructors initialize objects. Understanding default, parameterized, and copy constructors ensures flexibility in object creation. Overloading constructors and managing the object lifecycle using garbage collection are essential for efficient memory usage in Java programs.

3.4 Static vs Non-Static Features

In Java, the `static` keyword is used to define class-level variables, methods, and blocks. Understanding the difference between static and non-static features is critical for writing efficient and well-structured programs. Static features belong to the class itself, whereas non-static features belong to individual objects.

3.4.1 Static Variables and Methods

3.4.1.0.1 Static Variables (Class Variables) Static variables are shared across all instances of a class. They belong to the class rather than any specific object.

3.4.1.0.2 Example of Static Variables

```
class Counter {
  // Static variable
  static int count = 0;

  public Counter() {
    count++; // Increment static variable
  }

  public void displayCount() {
    System.out.println("Count: " + count);
  }
}

public class StaticVariableExample {
  public static void main(String[] args) {
    Counter c1 = new Counter();
    Counter c2 = new Counter();
    Counter c3 = new Counter();

    c1.displayCount(); // Output: 3
    c2.displayCount(); // Output: 3
    c3.displayCount(); // Output: 3
  }
}
```

3.4.1.0.3 Static Methods Static methods are methods that can be called without creating an instance of the class. They can access only static data and cannot use `this` or non-static members.

3.4.1.0.4 Example of Static Methods

```
class MathUtils {
  // Static method
  public static int square(int x) {
```



```
    return x * x;
  }
}

public class StaticMethodExample {
  public static void main(String[] args) {
    int result = MathUtils.square(5); // No object required
    System.out.println("Square: " + result);
  }
}
```

3.4.2 Static Blocks and Static Classes

3.4.2.0.1 Static Blocks Static blocks are used to initialize static variables. They are executed when the class is loaded into memory, before any object is created.

3.4.2.0.2 Example of Static Block

```
class StaticBlockExample {
  static int value;

  // Static block
  static {
    value = 10;
    System.out.println("Static block executed");
  }

  public static void displayValue() {
    System.out.println("Value: " + value);
  }
}

public class Main {
  public static void main(String[] args) {
    StaticBlockExample.displayValue();
  }
}
```

Output:

```
Static block executed
Value: 10
```

3.4.2.0.3 Static Classes (Nested Static Classes) A static class is a class declared inside another class with the `static` keyword. It can access only static members of the outer class.

3.4.2.0.4 Example of Static Class

```
class OuterClass {
  static int outerValue = 100;

  // Static nested class
  static class StaticNested {
    void display() {
      System.out.println("Outer Value: " + outerValue);
    }
  }
}

public class StaticClassExample {
  public static void main(String[] args) {
    OuterClass.StaticNested nested = new OuterClass.StaticNested();
    nested.display();
  }
}
```

Output:

```
Outer Value: 100
```

3.4.3 Static Context Limitations

3.4.3.0.1 Limitations of Static Context Static methods and blocks have the following limitations:

- They cannot access non-static (instance) variables or methods directly.
- The `this` keyword cannot be used because static methods do not belong to any instance.

3.4.3.0.2 Example: Static Method Cannot Access Non-Static Fields

```
class Example {
  int instanceVar = 10;

  // Static method
  static void display() {
    // System.out.println(instanceVar); // Error: Cannot access non-static variable
    System.out.println("Static method cannot access non-static fields directly.");
  }
}

public class Main {
  public static void main(String[] args) {
    Example.display();
  }
}
```


3.4.3.0.3 Resolving Static Context Limitations Non-static members can be accessed using an object reference within a static method:

```
class Example {
  int instanceVar = 10;

  static void display() {
    Example obj = new Example();
    System.out.println("Instance Variable: " + obj.instanceVar);
  }
}
```

3.4.4 Best Practices for Static Features

3.4.4.0.1 When to Use Static Variables

- Use static variables for data shared across all instances of a class (e.g., counters, constants).
- Avoid using static variables for storing instance-specific data.

3.4.4.0.2 When to Use Static Methods

- Use static methods for utility operations that do not depend on object state.
- Examples: Mathematical operations, helper methods, factory methods.

3.4.4.0.3 Avoid Excessive Use of Static Members Excessive use of static variables and methods can reduce flexibility and make code less modular. For instance, static members cannot be overridden in subclasses.

3.4.4.0.4 Avoid Using Static for Thread-Specific Data Static variables are shared across all instances, making them unsuitable for thread-specific data in multi-threaded applications.

3.4.4.0.5 Use Final with Static for Constants Declare constants as `static final` to ensure they remain unmodifiable and are shared across all instances:

```
class Constants {
  public static final double PI = 3.14159;
}
```


3.4.4.0.6 Example: Utility Class Design with Static Methods

Utility classes often contain only static methods and a private constructor to prevent instantiation:

```
class MathUtils {
  private MathUtils() {
    // Private constructor to prevent instantiation
  }

  public static int square(int x) {
    return x * x;
  }

  public static int cube(int x) {
    return x * x * x;
  }
}

public class Main {
  public static void main(String[] args) {
    System.out.println("Square: " + MathUtils.square(5));
    System.out.println("Cube: " + MathUtils.cube(3));
  }
}
```

Output:

```
Square: 25
Cube: 27
```

3.4.4.0.7 Summary of Static vs Non-Static Features

- **Static Features**:
  - Belong to the class, not the instance.
  - Used for shared data and utility methods.
  - Cannot directly access non-static members.
- **Non-Static Features**:
  - Belong to an instance of the class.
  - Allow unique data and behavior for each object.

Understanding the correct use of static and non-static members helps write clean, modular, and efficient Java programs.

3.5 Encapsulation and Access Modifiers

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It refers to the bundling of data (fields) and methods (behaviors) within a single unit (class) while restricting direct access to the internal state of the object. Access modifiers such as `public`, `private`, `protected`, and default (no modifier) control the visibility of class members and ensure data security.

3.5.1 Public, Private, Protected, and Default Access

3.5.1.0.1 What are Access Modifiers? Access modifiers control the visibility and accessibility of classes, fields, and methods. They define which parts of the program can access specific members.

3.5.1.0.2 Types of Access Modifiers Java provides four types of access modifiers:

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

3.5.1.0.3 Summary Table of Access Levels

Modifier	Class	Package	Subclass	World
`public`	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	No
default	Yes	Yes	No	No
`private`	Yes	No	No	No

3.5.2 Controlling Visibility of Class Members

3.5.2.0.1 Private Access Modifier The `private` modifier restricts access to fields and methods within the same class. It is commonly used for encapsulation.

```
class Person {
  // Private fields
  private String name;
  private int age;
```



```
  // Public getters and setters
  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public void setAge(int age) {
    this.age = age;
  }

  public int getAge() {
    return age;
  }
}

public class Main {
  public static void main(String[] args) {
    Person person = new Person();
    person.setName("Alice");
    person.setAge(30);

    System.out.println("Name: " + person.getName());
    System.out.println("Age: " + person.getAge());
  }
}
```

Output:

```
Name: Alice
Age: 30
```

3.5.2.0.2 Protected Access Modifier The `protected` modifier allows access to class members within the same package and in subclasses (even across different packages).

```
class Animal {
  protected void display() {
    System.out.println("This is a protected method.");
  }
}

class Dog extends Animal {
  public void show() {
    display(); // Accessible in subclass
  }
}

public class Main {
  public static void main(String[] args) {
    Dog dog = new Dog();
```



```
    dog.show();
  }
}
```

3.5.2.0.3 Default (No Modifier) Access Members with default access (no modifier) are accessible only within the same package.

```
class Example {
  void display() { // Default access
    System.out.println("Default access method.");
  }
}

public class Main {
  public static void main(String[] args) {
    Example example = new Example();
    example.display();
  }
}
```

3.5.3 Packages and Encapsulation

3.5.3.0.1 What are Packages? A package is a namespace that groups related classes and interfaces. It helps prevent naming conflicts and improves code organization.

3.5.3.0.2 Creating and Using a Package To declare a package, use the `package` keyword at the top of the file.

```
// File: mypackage/Person.java
package mypackage;

public class Person {
  private String name;

  public Person(String name) {
    this.name = name;
  }

  public void display() {
    System.out.println("Name: " + name);
  }
}
```

To use the class in another file:

```
// File: Main.java
import mypackage.Person;

public class Main {
  public static void main(String[] args) {
    Person p = new Person("Alice");
```



```
    p.display();
  }
}
```

3.5.4 Encapsulation Using Access Modifiers

3.5.4.0.1 Why Encapsulation Matters

Encapsulation provides:

- Controlled access to class fields through getters and setters.
- Improved data integrity by validating input values.
- Better maintainability and modularity of code.

3.5.4.0.2 Example: Validating Input with Encapsulation

Encapsulation ensures that invalid data cannot be assigned to class fields.

```
class BankAccount {
  private double balance;

  public void deposit(double amount) {
    if (amount > 0) {
      balance += amount;
    } else {
      System.out.println("Invalid deposit amount");
    }
  }

  public double getBalance() {
    return balance;
  }
}

public class Main {
  public static void main(String[] args) {
    BankAccount account = new BankAccount();
    account.deposit(500);
    account.deposit(-100); // Invalid
    System.out.println("Balance: " + account.getBalance());
  }
}
```

Output:

```
Invalid deposit amount
Balance: 500.0
```

3.5.5 Best Practices for Access Modifiers and Encapsulation

3.5.5.0.1 Use `private` for Fields

Always declare fields as `private` and expose them using public getters and setters.

3.5.5.0.2 Minimize the Use of `public` Restrict methods and fields to the smallest scope possible. Use `public` only when necessary.

3.5.5.0.3 Prefer `protected` for Inheritance Use `protected` for members that need to be accessible in subclasses but hidden from other classes.

3.5.5.0.4 Organize Code into Packages Group related classes into packages to improve modularity and maintainability.

3.5.5.0.5 Avoid Breaking Encapsulation Avoid exposing mutable internal fields. Instead, return copies or immutable versions of data.

```
class Data {
  private int[] values = {1, 2, 3};

  public int[] getValues() {
    return values.clone(); // Return a copy to prevent modification
  }
}
```

3.5.5.0.6 Summary of Access Modifiers and Encapsulation

- Use `private` to protect fields and expose methods for controlled access.
- Use `protected` for inheritance-based access.
- Use default access for package-level visibility.
- Encapsulation ensures data security, validation, and modularity.

By combining proper use of access modifiers with encapsulation, Java developers can create secure, maintainable, and well-organized programs.

3.6 Getters, Setters, and Mutators

In Java, getters, setters, and mutators play a significant role in accessing and modifying object fields while adhering to the principles of encapsulation. These methods control how fields are accessed and updated, ensuring data integrity and providing controlled visibility to class attributes.

3.6.1 Creating Getters and Setters

3.6.1.0.1 What are Getters and Setters?

- Getter Methods: Retrieve the value of private fields.
- Setter Methods: Update the value of private fields with optional validation.

Getters and setters allow class fields to remain private while still being accessible and modifiable indirectly.

3.6.1.0.2 Syntax for Getters and Setters The general convention for getters and setters is as follows:

```
class ClassName {
  private dataType fieldName;

  // Getter method
  public dataType getFieldName() {
    return fieldName;
  }

  // Setter method
  public void setFieldName(dataType value) {
    this.fieldName = value;
  }
}
```

3.6.1.0.3 Example of Getters and Setters

```
class Person {
  private String name;
  private int age;

  // Getter for name
  public String getName() {
    return name;
  }

  // Setter for name
  public void setName(String name) {
```



```
    this.name = name;
  }

  // Getter for age
  public int getAge() {
    return age;
  }

  // Setter for age with validation
  public void setAge(int age) {
    if (age > 0) {
      this.age = age;
    } else {
      System.out.println("Invalid age");
    }
  }
}

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

    // Using setters
    person.setName("Alice");
    person.setAge(25);

    // Using getters
    System.out.println("Name: " + person.getName());
    System.out.println("Age: " + person.getAge());
  }
}
```

Output:

```
Name: Alice
Age: 25
```

3.6.2 Immutable vs Mutable Objects

3.6.2.0.1 Mutable Objects Mutable objects allow fields to be updated after the object is created. By providing setters, the values of the fields can be modified.

```
class MutablePerson {
  private String name;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```



```
public class Main {
  public static void main(String[] args) {
    MutablePerson person = new MutablePerson();
    person.setName("John");
    System.out.println("Name: " + person.getName());

    person.setName("Alice"); // Field updated
    System.out.println("Updated Name: " + person.getName());
  }
}
```

3.6.2.0.2 Immutable Objects

Immutable objects do not allow their fields to be modified once they are initialized. To create an immutable object:

- Declare fields as `private` and `final`.
- Do not provide setter methods.
- Initialize fields using the constructor.

3.6.2.0.3 Example of an Immutable Class

```
final class ImmutablePerson {
  private final String name;
  private final int age;

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

  // Getter methods only
  public String getName() {
    return name;
  }

  public int getAge() {
    return age;
  }
}

public class Main {
  public static void main(String[] args) {
    ImmutablePerson person = new ImmutablePerson("Alice", 30);
    System.out.println("Name: " + person.getName());
    System.out.println("Age: " + person.getAge());
  }
}
```

Output:

```
Name: Alice
Age: 30
```


3.6.3 Bean Naming Conventions

3.6.3.0.1 What are Bean Naming Conventions? Java Bean naming conventions standardize the creation of getter and setter methods for class fields:

- Getter methods: Prefix with `get`, followed by the field name in PascalCase (e.g., `getName()`).
- Setter methods: Prefix with `set`, followed by the field name in PascalCase (e.g., `setName()`).
- Boolean getters: Prefix with `is` instead of `get`.

3.6.3.0.2 Example of Bean Naming Conventions

```
class Product {
  private String name;
  private double price;
  private boolean available;

  // Getter and setter for name
  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  // Getter and setter for price
  public double getPrice() {
    return price;
  }

  public void setPrice(double price) {
    this.price = price;
  }

  // Boolean getter uses 'is'
  public boolean isAvailable() {
    return available;
  }

  public void setAvailable(boolean available) {
    this.available = available;
  }
}

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

    product.setName("Laptop");
```



```
    product.setPrice(999.99);
    product.setAvailable(true);

    System.out.println("Product Name: " + product.getName());
    System.out.println("Price: \$" + product.getPrice());
    System.out.println("Available: " + product.isAvailable());
  }
}
```

Output:

```
Product Name: Laptop
Price: \$999.99
Available: true
```

3.6.4 Best Practices for Getters, Setters, and Mutators

3.6.4.0.1 Use Getters and Setters for Encapsulation Keep fields private and expose them through public getters and setters to control access and validation.

3.6.4.0.2 Validate Input in Setter Methods Add validation logic in setter methods to ensure only valid values are assigned.

```
public void setAge(int age) {
  if (age > 0) {
    this.age = age;
  } else {
    System.out.println("Age must be positive.");
  }
}
```

3.6.4.0.3 Avoid Unnecessary Setters for Immutable Classes For immutable objects, declare fields as `final` and do not provide setters.

3.6.4.0.4 Follow Bean Naming Conventions Adhere to the naming conventions for getters and setters to ensure compatibility with frameworks like Spring and tools like JSON serializers.

3.6.4.0.5 Use `is` for Boolean Getters Boolean fields should use the `is` prefix in getter methods instead of `get`.

3.6.4.0.6 Avoid Complex Logic in Getters and Setters Keep getters and setters simple. Avoid adding business logic that might confuse users of the class.

3.6.4.0.7 Summary of Getters, Setters, and Mutators

- Getters retrieve field values, while setters update them with validation.
- Immutable objects use final fields and exclude setters to prevent modification.
- Follow Java Bean naming conventions for better interoperability with frameworks.
- Encapsulation through getters and setters improves data security and maintainability.

By adhering to these principles, developers can create clean, secure, and well-structured Java classes.

3.7 Understanding the `this` and `super` Keywords

In Java, the `this` and `super` keywords play a vital role in managing object references and inheritance. The `this` keyword refers to the current instance of the class, while the `super` keyword is used to refer to members of the superclass. This chapter explains these keywords, their uses, and constructor chaining in detail.

3.7.1 The `this` Reference for Current Object

3.7.1.0.1 What is the `this` Keyword? The `this` keyword refers to the current object within a class. It is used to:

- Refer to the current instance variables when there is a name conflict.
- Call another constructor of the same class.
- Pass the current instance to a method or constructor.

3.7.1.0.2 Using `this` to Resolve Field and Parameter Name Conflicts When a parameter has the same name as an instance variable, the `this` keyword differentiates between the two.

```
class Person {
  private String name;

  // Constructor with parameter
  public Person(String name) {
    this.name = name; // 'this.name' refers to the instance variable
  }

  public void display() {
    System.out.println("Name: " + this.name);
  }
}

public class Main {
  public static void main(String[] args) {
    Person person = new Person("Alice");
    person.display(); // Output: Name: Alice
  }
}
```

3.7.1.0.3 Using `this` to Call a Method of the Current Object The `this` keyword can also call a method within the same class.


```
class Calculator {
  private int a, b;

  public Calculator(int a, int b) {
    this.a = a;
    this.b = b;
  }

  void add() {
    System.out.println("Sum: " + (a + b));
  }

  void display() {
    this.add(); // Calls the add() method
  }
}

public class Main {
  public static void main(String[] args) {
    Calculator calc = new Calculator(5, 10);
    calc.display(); // Output: Sum: 15
  }
}
```

3.7.2 The super Keyword for Superclass Members

3.7.2.0.1 What is the super Keyword? The `super` keyword is used to refer to members of the superclass (parent class) in a subclass. It is commonly used to:

- Access superclass fields and methods.
- Call the superclass constructor.

3.7.2.0.2 Using super to Access Superclass Fields If a subclass has a field with the same name as a field in the superclass, the `super` keyword can differentiate between them.

```
class Parent {
  String name = "Parent";
}

class Child extends Parent {
  String name = "Child";

  void display() {
    System.out.println("Subclass name: " + name);
    System.out.println("Superclass name: " + super.name); // Refers to Parent's 'name'
  }
}

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



```
    Child child = new Child();
    child.display();
  }
}
```

3.7.2.0.3 Using `super` to Call Superclass Methods

The `super` keyword can call a method from the superclass that is overridden in the subclass.

```
class Parent {
  void display() {
    System.out.println("Display method in Parent");
  }
}

class Child extends Parent {
  void display() {
    super.display(); // Call superclass method
    System.out.println("Display method in Child");
  }
}

public class Main {
  public static void main(String[] args) {
    Child child = new Child();
    child.display();
  }
}
```

Output:

```
Display method in Parent
Display method in Child
```

3.7.3 Chaining Constructors Using `this()` and `super()`

3.7.3.0.1 What is Constructor Chaining?

Constructor chaining is the process of calling one constructor from another. It can occur within the same class using `this()` or between superclass and subclass constructors using `super()`.

3.7.3.0.2 Using `this()` to Call a Constructor in the Same Class

The `this()` keyword calls another constructor in the same class.

```
class Person {
  String name;
  int age;

  // Default constructor
  public Person() {
    this("Unknown", 0); // Calls the parameterized constructor
```



```
  }

  // Parameterized constructor
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public void display() {
    System.out.println("Name: " + name + ", Age: " + age);
  }
}

public class Main {
  public static void main(String[] args) {
    Person p1 = new Person(); // Calls default constructor
    Person p2 = new Person("Alice", 30); // Calls parameterized constructor

    p1.display();
    p2.display();
  }
}
```

Output:

```
Name: Unknown, Age: 0
Name: Alice, Age: 30
```

3.7.3.0.3 Using `super()` to Call the Superclass Constructor The `super()` keyword calls the superclass constructor. It must be the first statement in the subclass constructor.

```
class Parent {
  String name;

  Parent(String name) {
    this.name = name;
    System.out.println("Parent Constructor: " + name);
  }
}

class Child extends Parent {
  Child(String name) {
    super(name); // Calls Parent's constructor
    System.out.println("Child Constructor: " + name);
  }
}

public class Main {
  public static void main(String[] args) {
    Child child = new Child("Alice");
  }
}
```


Output:

```
Parent Constructor: Alice
Child Constructor: Alice
```

3.7.4 Best Practices for Using `this` and `super`

3.7.4.0.1 Best Practices for `this`

- Use `this` to resolve naming conflicts between fields and parameters.
- Use `this()` for constructor chaining to avoid code duplication.
- Avoid overusing `this` where it is unnecessary.

3.7.4.0.2 Best Practices for `super`

- Use `super` to explicitly access superclass members when overridden.
- Always place `super()` as the first statement in the subclass constructor.
- Avoid calling `super` unless necessary to keep the code clean and readable.

3.7.4.0.3 Combining `this` and `super` You cannot use both `this()` and `super()` in the same constructor, as both must be the first statement.

3.7.4.0.4 Summary of `this` and `super`

- `this` refers to the current instance and is used to resolve conflicts, call methods, or chain constructors.
- `super` refers to the superclass and is used to access superclass fields, methods, and constructors.
- Constructor chaining using `this()` and `super()` improves code reuse and clarity.

By understanding and applying the `this` and `super` keywords effectively, developers can write clean, modular, and maintainable Java programs.

3.8 Concepts and Implementation of Inheritance

Inheritance is a core principle of Object-Oriented Programming (OOP) that allows a class to acquire the properties and behaviors of another class. This promotes code reuse, reduces redundancy, and enables hierarchical relationships among classes.

3.8.1 Single and Multi-Level Inheritance

3.8.1.0.1 What is Inheritance? Inheritance allows one class (the subclass or child class) to inherit fields and methods from another class (the superclass or parent class). The relationship between the classes is represented using the `extends` keyword.

3.8.1.0.2 Single-Level Inheritance In single-level inheritance, a subclass inherits directly from a single parent class.

3.8.1.0.3 Example of Single-Level Inheritance

```
class Parent {
  void display() {
    System.out.println("This is the Parent class.");
  }
}

class Child extends Parent { // Single-level inheritance
  void show() {
    System.out.println("This is the Child class.");
  }
}

public class Main {
  public static void main(String[] args) {
    Child child = new Child();
    child.display(); // Inherited from Parent
    child.show();    // Child class method
  }
}
```

Output:

```
This is the Parent class.
This is the Child class.
```

3.8.1.0.4 Multi-Level Inheritance In multi-level inheritance, a class inherits from a child class, creating a chain of inheritance.

3.8.1.0.5 Example of Multi-Level Inheritance

```
class GrandParent {
  void showGrandParent() {
    System.out.println("This is the Grandparent class.");
  }
}

class Parent extends GrandParent {
  void showParent() {
    System.out.println("This is the Parent class.");
  }
}

class Child extends Parent {
  void showChild() {
    System.out.println("This is the Child class.");
  }
}

public class Main {
  public static void main(String[] args) {
    Child child = new Child();
    child.showGrandParent(); // Inherited from GrandParent
    child.showParent();      // Inherited from Parent
    child.showChild();       // Child class method
  }
}
```

Output:

```
This is the Grandparent class.
This is the Parent class.
This is the Child class.
```

3.8.2 Method Overriding and Using `super()`

3.8.2.0.1 What is Method Overriding? Method overriding occurs when a subclass provides a specific implementation for a method that already exists in the superclass. Overridden methods must have the same name, return type, and parameters.

3.8.2.0.2 Rules for Method Overriding

- The method in the subclass must have the same signature as the one in the superclass.
- The overriding method cannot have a weaker access modifier.
- Use the `@Override` annotation to ensure proper overriding.

3.8.2.0.3 Example of Method Overriding

```
class Parent {
  void display() {
    System.out.println("This is the Parent class method.");
  }
}

class Child extends Parent {
  @Override
  void display() {
    System.out.println("This is the Child class method.");
  }
}

public class Main {
  public static void main(String[] args) {
    Parent obj = new Child(); // Polymorphic behavior
    obj.display(); // Calls Child's overridden method
  }
}
```

Output:

```
This is the Child class method.
```

3.8.2.0.4 Using `super()` to Call Superclass Methods The `super` keyword is used to call the superclass version of a method from the subclass.

```
class Parent {
  void display() {
    System.out.println("This is the Parent class method.");
  }
}

class Child extends Parent {
  @Override
  void display() {
    super.display(); // Call superclass method
    System.out.println("This is the Child class method.");
  }
}

public class Main {
  public static void main(String[] args) {
    Child child = new Child();
    child.display();
  }
}
```

Output:

```
This is the Parent class method.
This is the Child class method.
```


3.8.3 Is-A vs Has-A Relationship

3.8.3.0.1 Understanding Is-A Relationship (Inheritance) The **Is-A** relationship represents inheritance. A subclass is a type of its superclass.

- Example: A **Dog Is-A Animal**.
- Implemented using the `extends` keyword.

3.8.3.0.2 Example of Is-A Relationship

```
class Animal {
  void eat() {
    System.out.println("This animal eats food.");
  }
}

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

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

Output:

```
This animal eats food.
The dog barks.
```

3.8.3.0.3 Understanding Has-A Relationship (Composition) The **Has-A** relationship represents composition, where one class contains a reference to another class.

- Example: A **Car Has-A Engine**.
- Implemented by including an instance of one class within another.

3.8.3.0.4 Example of Has-A Relationship

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

class Car {
  // Has-A relationship
  Engine engine = new Engine();

  void startCar() {
    engine.start(); // Using Engine's method
    System.out.println("Car is ready to drive.");
  }
}

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

Output:

```
Engine is starting...
Car is ready to drive.
```

3.8.4 Key Differences: Is-A vs Has-A Relationship

- Is-A: Represents inheritance; one class **extends** another class.
- Has-A: Represents composition; one class **contains** another class as a field.

3.8.4.0.1 Comparison Table

Relationship	Implementation	Example
Is-A	`extends`	Dog Is-A Animal
Has-A	Composition (fields)	Car Has-A Engine

3.8.5 Best Practices for Inheritance and Composition

3.8.5.0.1 Best Practices for Inheritance (Is-A)

- Use inheritance only when there is a clear Is-A relationship.
- Avoid deep inheritance hierarchies as they make code harder to maintain.
- Use `super` to reuse superclass logic where appropriate.

3.8.5.0.2 Best Practices for Composition (Has-A)

- Prefer composition over inheritance for better flexibility.
- Use composition to avoid tight coupling between classes.
- Encapsulate class fields to maintain data integrity.

3.8.5.0.3 Summary of Inheritance and Relationships

- Single and multi-level inheritance allow classes to reuse logic and maintain a hierarchy.
- Method overriding enables subclasses to provide specific implementations for inherited methods.
- The **Is-A** relationship represents inheritance, while the **Has-A** relationship represents composition.
- Use inheritance for clear hierarchies and composition for flexible, modular designs.

By combining inheritance, composition, and proper use of `super`, developers can create maintainable and extensible object-oriented designs.

3.9 Method Overriding and Dynamic Binding

Method overriding and dynamic binding are key components of achieving polymorphism in Java. They allow a subclass to provide a specific implementation for a method already defined in its superclass, enabling flexibility and runtime behavior determination.

3.9.1 Compile-Time vs Run-Time Polymorphism

3.9.1.0.1 What is Polymorphism? Polymorphism means "many forms." It allows objects to be treated as instances of their parent class while exhibiting different behavior based on their actual implementation.

3.9.1.0.2 Compile-Time Polymorphism (Method Overloading) Compile-time polymorphism occurs when the method to be called is resolved at compile time. This is achieved through method overloading, where multiple methods share the same name but differ in parameters.

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

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

public class Main {
  public static void main(String[] args) {
    Calculator calc = new Calculator();
    System.out.println(calc.add(5, 10));      // Calls int version
    System.out.println(calc.add(2.5, 3.5));   // Calls double version
  }
}
```

Output:

```
15
6.0
```

3.9.1.0.3 Run-Time Polymorphism (Method Overriding) Run-time polymorphism occurs when the method to be called is resolved during program execution. This is achieved through method overriding.

- A subclass provides a specific implementation of a method defined in its superclass.
- The method call is determined dynamically at runtime.

3.9.2 Dynamic Method Dispatch

3.9.2.0.1 What is Dynamic Method Dispatch? Dynamic Method Dispatch (or runtime polymorphism) refers to the process where a method call on a superclass reference is resolved to the appropriate subclass implementation at runtime.

3.9.2.0.2 Example of Method Overriding and Dynamic Binding

```
class Animal {
  void sound() {
    System.out.println("Animals make sounds");
  }
}

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

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

public class Main {
  public static void main(String[] args) {
    Animal animal;

    animal = new Dog(); // Dynamic binding
    animal.sound();     // Calls Dog's sound()

    animal = new Cat(); // Dynamic binding
    animal.sound();     // Calls Cat's sound()
  }
}
```

Output:

```
Dog barks
Cat meows
```


3.9.2.0.3 How Dynamic Binding Works

At runtime:

- The reference variable type (e.g., `Animal`) determines what methods can be called.
- The actual object (e.g., `Dog` or `Cat`) determines which method implementation is executed.

3.9.3 Polymorphic Behavior with Abstract References

3.9.3.0.1 Using Abstract Classes for Polymorphism

Abstract classes allow us to define a common behavior for subclasses while letting subclasses provide their own implementation.

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

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

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

public class Main {
  public static void main(String[] args) {
    Shape shape;

    shape = new Circle();
    shape.draw(); // Polymorphic behavior

    shape = new Rectangle();
    shape.draw(); // Polymorphic behavior
  }
}
```

Output:

```
Drawing a Circle
Drawing a Rectangle
```


3.9.3.0.2 Using Interfaces for Polymorphic Behavior

Interfaces can also be used to achieve polymorphism because they allow different classes to implement the same set of methods.

```
interface Animal {
  void sound();
}

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

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

public class Main {
  public static void main(String[] args) {
    Animal animal;

    animal = new Dog();
    animal.sound(); // Polymorphic behavior

    animal = new Cat();
    animal.sound(); // Polymorphic behavior
  }
}
```

Output:

```
Dog barks
Cat meows
```

3.9.4 Comparison of Compile-Time and Run-Time Polymorphism

3.9.4.0.1 Differences Between Compile-Time and Run-Time Polymorphism

Aspect	Compile-Time Polymorphism	Run-Time Polymorphism
Achieved By	Method Overloading	Method Overriding
Resolution Time	Compile-time	Run-time
Flexibility	Less flexible	More flexible
Binding Type	Static Binding	Dynamic Binding
Example	Method Overloading	Method Overriding

3.9.5 Best Practices for Polymorphism

3.9.5.0.1 Use Method Overriding for Dynamic Behavior Override methods in subclasses to provide specific implementations for general behaviors defined in the superclass.

3.9.5.0.2 Always Use `@Override` Annotation The `@Override` annotation ensures that a method is correctly overriding a superclass method.

```
@Override
void sound() {
  System.out.println("Custom sound");
}
```

3.9.5.0.3 Use Abstract Classes or Interfaces for Polymorphism Abstract classes and interfaces allow you to define a common contract for different implementations.

3.9.5.0.4 Avoid Overloading Confusion Ensure method overloading is meaningful and does not cause ambiguity for the developer.

3.9.5.0.5 Prefer Base Class References for Flexibility Always use base class or interface references to achieve polymorphism, which makes your code extensible.

3.9.5.0.6 Summary of Method Overriding and Polymorphism

- Compile-Time Polymorphism: Achieved using method overloading; resolved at compile time.
- Run-Time Polymorphism: Achieved using method overriding and dynamic binding; resolved at runtime.
- Dynamic Method Dispatch: Enables a superclass reference to determine the correct method implementation at runtime.
- Polymorphism with Abstract Classes/Interfaces: Allows different implementations while adhering to a common interface.

Polymorphism is a powerful concept that simplifies program design, improves code reusability, and allows dynamic behavior during runtime.

3.10 Abstract Classes and Interfaces

Abstract classes and interfaces are critical components of Java's object-oriented programming (OOP) model. They enable abstraction, which hides implementation details and exposes only essential behavior. While abstract classes provide a partial implementation, interfaces define a contract that must be implemented by classes.

3.10.1 When to Use Abstract Classes

3.10.1.0.1 What is an Abstract Class? An abstract class is a class that cannot be instantiated. It may contain both abstract methods (without implementations) and concrete methods (with implementations). It is designed to be extended by subclasses that provide specific implementations.

3.10.1.0.2 Characteristics of Abstract Classes

- Declared using the `abstract` keyword.
- Can contain abstract and non-abstract (concrete) methods.
- May have instance variables, constructors, and static methods.
- Cannot be instantiated directly.

3.10.1.0.3 Syntax for Abstract Classes

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

  // Concrete method
  void display() {
    System.out.println("This is a shape.");
  }
}
```

3.10.1.0.4 Example: Abstract Class Implementation

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

  void display() { // Concrete method
    System.out.println("This is a shape.");
  }
}
```



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

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

public class Main {
  public static void main(String[] args) {
    Shape shape1 = new Circle();
    shape1.draw(); // Outputs: Drawing a Circle
    shape1.display();

    Shape shape2 = new Rectangle();
    shape2.draw(); // Outputs: Drawing a Rectangle
  }
}
```

Output:

```
Drawing a Circle
This is a shape.
Drawing a Rectangle
```

3.10.1.0.5 When to Use Abstract Classes

- When you want to provide a common base class for related subclasses.
- When you need to share code (concrete methods) across subclasses.
- When you expect subclasses to provide their own implementations for specific methods (abstract methods).

3.10.2 Defining and Implementing Interfaces

3.10.2.0.1 What is an Interface? An interface is a reference type that defines a contract of methods that a class must implement. Interfaces support multiple inheritance and promote loose coupling.

3.10.2.0.2 Characteristics of Interfaces

- Declared using the `interface` keyword.
- All methods in an interface are `public` and `abstract` by default.
- Fields in an interface are implicitly `public`, `static`, and `final`.
- A class can implement multiple interfaces.

3.10.2.0.3 Syntax for Interfaces

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

3.10.2.0.4 Implementing an Interface A class implements an interface using the `implements` keyword and must provide implementations for all abstract methods.

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

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

class Cat implements Animal {
  @Override
  public void sound() {
    System.out.println("Cat meows");
  }
}

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

    Animal cat = new Cat();
    cat.sound(); // Outputs: Cat meows
  }
}
```

Output:

```
Dog barks
Cat meows
```


3.10.2.0.5 When to Use Interfaces

- When you want to define a common behavior across unrelated classes.
- To achieve multiple inheritance in Java.
- To define a contract that must be followed by implementing classes.

3.10.3 Default and Static Methods in Interfaces

3.10.3.0.1 Default Methods in Interfaces (Java 8+) Default methods allow an interface to provide a default implementation for a method. Classes implementing the interface can either use the default implementation or override it.

3.10.3.0.2 Example of Default Methods

```
interface Animal {
  void sound();

  // Default method
  default void sleep() {
    System.out.println("Animals sleep");
  }
}

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

public class Main {
  public static void main(String[] args) {
    Dog dog = new Dog();
    dog.sound(); // Calls overridden method
    dog.sleep(); // Calls default method
  }
}
```

Output:

```
Dog barks
Animals sleep
```

3.10.3.0.3 Static Methods in Interfaces (Java 8+) Static methods in interfaces are similar to static methods in classes. They can be called directly using the interface name.

3.10.3.0.4 Example of Static Methods

```
interface MathOperations {
  static int square(int x) {
    return x * x;
  }
}

public class Main {
  public static void main(String[] args) {
    System.out.println("Square of 5: " + MathOperations.square(5));
  }
}
```

Output:

```
Square of 5: 25
```

3.10.4 Key Differences Between Abstract Classes and Interfaces

3.10.4.0.1 Comparison Table

Aspect	Abstract Class	Interface
Keyword	`abstract`	`interface`
Methods	Can have abstract and concrete methods	Only abstract methods (Java 7) Default and static methods (Java 8+)
Variables	Can have instance variables	Only `public static final` variables
Inheritance	Single inheritance only	Multiple inheritance supported
Constructors	Can have constructors	Cannot have constructors
Access Modifiers	Can use all access modifiers	All methods are `public` by default

3.10.5 Best Practices for Abstract Classes and Interfaces

3.10.5.0.1 Best Practices for Abstract Classes

- Use abstract classes when there is a clear hierarchy and shared code among subclasses.
- Define abstract methods only when subclasses must provide an implementation.
- Avoid deep inheritance chains to reduce complexity.

3.10.5.0.2 Best Practices for Interfaces

- Use interfaces to define common behavior for unrelated classes.
- Prefer interfaces when multiple inheritance of behavior is needed.
- Use default and static methods sparingly to avoid bloating interfaces.

3.10.5.0.3 Summary of Abstract Classes and Interfaces

- Abstract classes are partially implemented classes that can contain abstract and concrete methods.
- Interfaces define a contract that must be implemented by classes.
- Default and static methods in interfaces (introduced in Java 8) provide flexibility and shared behavior.
- Choose abstract classes for shared code and interfaces for multiple inheritance of behavior.

By understanding and applying abstract classes and interfaces, developers can write clean, modular, and maintainable Java programs.

3.11 Composition and Aggregation

Composition and aggregation are two key concepts in object-oriented programming (OOP) that help define relationships between classes. Both are forms of the "Has-A" relationship, where one class contains an instance of another class. Understanding these concepts is essential for creating modular, reusable, and maintainable code.

3.11.1 Understanding Aggregation and Composition

3.11.1.0.1 What is Aggregation? Aggregation represents a weak "Has-A" relationship where one class contains a reference to another class, but the lifecycle of the referenced object is independent of the container class.

- Aggregation allows an object to share ownership of another object.
- If the container object is destroyed, the referenced object can still exist.

3.11.1.0.2 What is Composition? Composition represents a strong "Has-A" relationship where one class contains an instance of another class, and the lifecycle of the contained object depends on the container class.

- If the container object is destroyed, the contained object is also destroyed.
- Composition is a stronger form of association compared to aggregation.

3.11.2 Implementing Has-A Relationships

3.11.2.0.1 Example of Aggregation Aggregation allows one class to "use" another class while maintaining separate lifecycles.

```
class Address {
  String city, state;

  public Address(String city, String state) {
    this.city = city;
    this.state = state;
  }

  public void display() {
    System.out.println("City: " + city + ", State: " + state);
  }
}

class Employee {
  String name;
```



```
  Address address; // Aggregation: Employee "Has-A" Address

  public Employee(String name, Address address) {
    this.name = name;
    this.address = address;
  }

  public void displayInfo() {
    System.out.println("Employee Name: " + name);
    address.display(); // Use Address class
  }
}

public class Main {
  public static void main(String[] args) {
    Address addr = new Address("New York", "NY");
    Employee emp = new Employee("John", addr);

    emp.displayInfo();
  }
}
```

Output:

```
Employee Name: John
City: New York, State: NY
```

3.11.2.0.2 Explanation of Aggregation: In this example:

- The `Employee` class "has-a" relationship with the `Address` class.
- The lifecycle of the `Address` object is independent of the `Employee` object.

3.11.2.0.3 Example of Composition Composition enforces ownership, where the contained object cannot exist without the container object.

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

class Car {
  private Engine engine; // Composition: Car "Has-A" Engine

  public Car() {
    engine = new Engine(); // Engine is created when Car is created
  }

  public void startCar() {
    engine.start();
    System.out.println("Car is ready to drive.");
```



```
  }
}

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

Output:

```
Engine is starting...
Car is ready to drive.
```

3.11.2.0.4 Explanation of Composition:

In this example:

- The `Car` class "has-a" relationship with the `Engine` class.
- The `Engine` object is created within the `Car` class, and its lifecycle is dependent on the `Car` object.
- When the `Car` object is destroyed, the `Engine` object is also destroyed.

3.11.3 Comparing Composition and Inheritance

3.11.3.0.1 Composition vs Inheritance

Both composition and inheritance promote code reuse but serve different purposes:

Aspect	Composition	Inheritance
Relationship	"Has-A" relationship	"Is-A" relationship
Coupling	Loosely coupled	Tightly coupled
Flexibility	More flexible (can change behavior)	Less flexible (fixed inheritance)
Reuse	Uses objects to reuse functionality	Reuses behavior via subclassing
Maintenance	Easier to maintain	Can cause issues with deep hierarchies

3.11.3.0.2 Choosing Between Composition and Inheritance

- Use composition when you want flexibility and decoupling between classes.
- Use inheritance when there is a clear hierarchical relationship (**Is-A** relationship).

3.11.3.0.3 Example: Composition vs Inheritance

```java
// Composition Example
class Engine {
  void start() {
    System.out.println("Engine is starting...");
  }
}

class Car {
  Engine engine = new Engine(); // "Has-A" relationship

  void startCar() {
    engine.start();
    System.out.println("Car is moving.");
  }
}

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

class Bike extends Vehicle { // "Is-A" relationship
  @Override
  void move() {
    System.out.println("Bike is moving...");
  }
}

public class Main {
  public static void main(String[] args) {
    // Composition
    Car car = new Car();
    car.startCar();

    // Inheritance
    Bike bike = new Bike();
    bike.move();
  }
}
```

Output:

```
Engine is starting...
Car is moving.
Bike is moving...
```

3.11.4 Best Practices for Composition and Aggregation

3.11.4.0.1 Best Practices for Composition

- Prefer composition over inheritance when flexibility is required.
- Use composition to reuse behavior without tightly coupling classes.
- Avoid creating deep inheritance hierarchies by favoring composition.

3.11.4.0.2 Best Practices for Aggregation

- Use aggregation when the contained object can exist independently of the container.
- Ensure relationships are clear and reflect real-world modeling.
- Avoid unnecessary references to unrelated objects.

3.11.4.0.3 Summary of Composition and Aggregation

- Aggregation represents a weak "Has-A" relationship where objects can exist independently.
- Composition represents a strong "Has-A" relationship where the lifecycle of contained objects depends on the container.
- Use composition for flexibility and when you want to avoid the limitations of inheritance.
- Carefully choose between aggregation, composition, and inheritance based on the problem structure.

By understanding and correctly implementing aggregation and composition, developers can design flexible, maintainable, and modular Java programs.

3.12 Inner Classes and Anonymous Classes

In Java, inner classes and anonymous classes allow developers to logically group classes and provide a more concise way to create class instances. They improve code organization, readability, and are often used to implement event handling or simplify code where creating a separate class would be overkill.

3.12.1 Static and Non-Static Inner Classes

3.12.1.0.1 What are Inner Classes? An inner class is a class defined within another class. Inner classes provide better encapsulation and can access the fields and methods of their enclosing class. There are two main types:

- Static Inner Classes: Nested classes with the `static` keyword.
- Non-Static Inner Classes: Classes that are instance-level and tied to the enclosing class's object.

3.12.1.0.2 Static Inner Classes Static inner classes can be instantiated without an instance of the outer class. They can only access static members of the enclosing class.

3.12.1.0.3 Example of Static Inner Class

```
class OuterClass {
  static String staticField = "Static Field in OuterClass";

  // Static Inner Class
  static class StaticInner {
    void display() {
      System.out.println("Accessing: " + staticField);
    }
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass.StaticInner inner = new OuterClass.StaticInner();
    inner.display();
  }
}
```

Output:

```
Accessing: Static Field in OuterClass
```


3.12.1.0.4 Non-Static Inner Classes

Non-static inner classes are tied to an instance of the outer class. They can access both static and non-static members of the outer class.

3.12.1.0.5 Example of Non-Static Inner Class

```
class OuterClass {
  private String instanceField = "Instance Field in OuterClass";

  // Non-Static Inner Class
  class Inner {
    void display() {
      System.out.println("Accessing: " + instanceField);
    }
  }
}

public class Main {
  public static void main(String[] args) {
    OuterClass outer = new OuterClass();
    OuterClass.Inner inner = outer.new Inner(); // Inner class tied to OuterClass instance
    inner.display();
  }
}
```

Output:

```
Accessing: Instance Field in OuterClass
```

3.12.2 Local Inner Classes

3.12.2.0.1 What are Local Inner Classes?

Local inner classes are defined within a method or block. They are accessible only within the scope of that method or block.

3.12.2.0.2 Example of Local Inner Class

```
class OuterClass {
  void display() {
    // Local Inner Class
    class LocalInner {
      void printMessage() {
        System.out.println("Inside Local Inner Class");
      }
    }
    LocalInner inner = new LocalInner();
    inner.printMessage();
  }
}
```



```
public class Main {
  public static void main(String[] args) {
    OuterClass outer = new OuterClass();
    outer.display();
  }
}
```

Output:

```
Inside Local Inner Class
```

3.12.2.0.3 Key Features of Local Inner Classes:

- They are defined inside a method or block.
- They cannot have access modifiers (`public`, `private`, etc.).
- They can access local variables of the enclosing method only if those variables are `final` or effectively final.

3.12.3 Anonymous Classes and Functional Usage

3.12.3.0.1 What is an Anonymous Class? An anonymous class is a class without a name that is created as part of a single expression. It is used when a class needs to be instantiated only once, typically for:

- Event handling
- Simplifying code where a full class definition is unnecessary

3.12.3.0.2 Syntax of an Anonymous Class Anonymous classes are defined using the following syntax:

```
interface Greeting {
  void sayHello();
}

public class Main {
  public static void main(String[] args) {
    Greeting greeting = new Greeting() {
      @Override
      public void sayHello() {
        System.out.println("Hello from Anonymous Class");
      }
    };
    greeting.sayHello();
  }
}
```


Output:

```
Hello from Anonymous Class
```

3.12.3.0.3 Example: Anonymous Class for Thread Creation

Anonymous classes are often used to implement interfaces or extend classes for quick use.

```
public class Main {
  public static void main(String[] args) {
    // Anonymous class implementing Runnable
    Runnable task = new Runnable() {
      @Override
      public void run() {
        System.out.println("Task executed by anonymous class");
      }
    };

    Thread thread = new Thread(task);
    thread.start();
  }
}
```

Output:

```
Task executed by anonymous class
```

3.12.4 Functional Usage with Anonymous Classes

3.12.4.0.1 Functional Interfaces and Lambdas (Java 8+)

Since Java 8, anonymous classes can often be replaced with lambda expressions for functional interfaces, simplifying the code further.

3.12.4.0.2 Example: Using Lambda Expression Instead of Anonymous Class

```
interface Greeting {
  void sayHello();
}

public class Main {
  public static void main(String[] args) {
    // Lambda replacing anonymous class
    Greeting greeting = () -> System.out.println("Hello using Lambda");
    greeting.sayHello();
  }
}
```

Output:

```
Hello using Lambda
```


3.12.5 Comparing Different Types of Inner Classes

3.12.5.0.1 Summary Table for Inner Classes

Type of Inner Class	Definition	Access
Static Inner Class	Declared with the `static` keyword	Can access only static me
Non-Static Inner Class	Instance-level class inside another class	Can access all outer class
Local Inner Class	Defined inside a method or block	Visible only within the me
Anonymous Class	Inline implementation of class/interface	Exists for one-time use

3.12.6 Best Practices for Inner and Anonymous Classes

3.12.6.0.1 Best Practices

- Use static inner classes when the inner class does not need access to outer class instance members.
- Use non-static inner classes for tightly coupled functionality that relies on the outer class.
- Use local inner classes sparingly for small, specific tasks within methods.
- Prefer anonymous classes for one-time use, but consider lambda expressions when applicable.

3.12.6.0.2 Summary of Inner Classes and Anonymous Classes

- Inner classes allow logical grouping of classes and improve encapsulation.
- Static inner classes are independent of outer class instances, while non-static inner classes are tied to outer class objects.
- Local inner classes are defined within methods and have limited scope.
- Anonymous classes simplify code for short-term implementations, but lambdas are preferred for functional interfaces.

By mastering inner classes and anonymous classes, developers can write more modular and concise Java programs while improving code readability.

3.13 Exception Handling and Error Management

Exception handling in Java provides a robust mechanism for managing runtime errors, ensuring that the program can recover gracefully without crashing. Java's exception-handling model allows developers to detect, handle, and propagate errors efficiently.

3.13.1 Types of Exceptions: Checked, Unchecked, and Errors

3.13.1.0.1 What is an Exception? An exception is an abnormal condition that occurs during the execution of a program, disrupting its normal flow. Exceptions are objects in Java that are derived from the `Throwable` class.

3.13.1.0.2 Types of Exceptions and Errors

- **Checked Exceptions**: Exceptions that are checked at compile-time.
- **Unchecked Exceptions (Runtime Exceptions)**: Exceptions that occur at runtime and are not checked at compile-time.
- **Errors**: Serious issues that a program cannot handle (e.g., `OutOfMemoryError`).

3.13.1.0.3 Hierarchy of Exceptions and Errors In Java, the root class for handling exceptional conditions is `Throwable`. It serves as the superclass for two primary categories: `Exception` and `Error`. The `Exception` class represents conditions that applications might want to catch and handle. Exceptions are further classified into *checked exceptions* and *unchecked exceptions*. Checked exceptions, such as `IOException` and `SQLException`, must be declared in the method signature or explicitly handled with a `try-catch` block. Unchecked exceptions are subclasses of `RuntimeException` and include common errors like `NullPointerException`, `ArrayIndexOutOfBoundsExce` and `ArithmeticException`.

On the other hand, the `Error` class represents critical problems that generally cannot be recovered from and indicate serious issues within the JVM. Examples include `OutOfMemoryError`, which occurs when the Java Virtual Machine cannot allocate memory, `StackOverflowError`, caused by excessive deep recursion, and `VirtualMachineError`, signaling underlying JVM failures. Unlike exceptions, errors are typically not caught or handled by applications, as they usually represent fatal conditions.

3.13.1.0.4 Example of Checked Exception

Checked exceptions must be handled using `try-catch` or declared using `throws`.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class CheckedExceptionExample {
  public static void main(String[] args) {
    try {
      File file = new File("nonexistent.txt");
      Scanner scanner = new Scanner(file); // Throws FileNotFoundException
    } catch (FileNotFoundException e) {
      System.out.println("File not found: " + e.getMessage());
    }
  }
}
```

3.13.1.0.5 Example of Unchecked Exception

Unchecked exceptions are not required to be handled.

```
public class UncheckedExceptionExample {
  public static void main(String[] args) {
    int[] numbers = {1, 2, 3};
    System.out.println(numbers[5]); // Throws ArrayIndexOutOfBoundsException
  }
}
```

3.13.1.0.6 Example of Error

Errors represent severe issues that are beyond the application's control.

```
public class ErrorExample {
  public static void main(String[] args) {
    recurse(); // Causes StackOverflowError
  }

  static void recurse() {
    recurse();
  }
}
```

3.13.2 `try`, `catch`, `finally`, and `throw`

3.13.2.0.1 Exception Handling Mechanism

Java provides the following keywords for exception handling:

- `try`: Block of code where exceptions can occur.
- `catch`: Handles the exception.

- `finally`: Executes cleanup code, regardless of exceptions.
- `throw`: Explicitly throws an exception.
- `throws`: Declares exceptions in the method signature.

3.13.2.0.2 Example of `try`, `catch`, and `finally`

```
public class TryCatchFinallyExample {
  public static void main(String[] args) {
    try {
      int result = 10 / 0; // Throws ArithmeticException
    } catch (ArithmeticException e) {
      System.out.println("Exception caught: " + e.getMessage());
    } finally {
      System.out.println("This block always executes.");
    }
  }
}
```

Output:

```
Exception caught: / by zero
This block always executes.
```

3.13.2.0.3 Using the `throw` Keyword

The `throw` keyword is used to explicitly throw an exception.

```
public class ThrowExample {
  public static void validateAge(int age) {
    if (age < 18) {
      throw new IllegalArgumentException("Age must be 18 or older");
    }
    System.out.println("Valid age: " + age);
  }

  public static void main(String[] args) {
    try {
      validateAge(16);
    } catch (IllegalArgumentException e) {
      System.out.println("Error: " + e.getMessage());
    }
  }
}
```

Output:

```
Error: Age must be 18 or older
```


3.13.3 Custom Exceptions and Exception Chaining

3.13.3.0.1 Creating Custom Exceptions Custom exceptions allow you to create meaningful, application-specific exceptions by extending the `Exception` or `RuntimeException` class.

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

public class CustomExceptionExample {
  public static void validateAge(int age) throws InvalidAgeException {
    if (age < 18) {
      throw new InvalidAgeException("Custom Exception: Invalid age");
    }
    System.out.println("Age is valid");
  }

  public static void main(String[] args) {
    try {
      validateAge(15);
    } catch (InvalidAgeException e) {
      System.out.println(e.getMessage());
    }
  }
}
```

Output:

```
Custom Exception: Invalid age
```

3.13.3.0.2 Exception Chaining Exception chaining allows you to wrap one exception inside another, preserving the root cause.

```
public class ExceptionChainingExample {
  public static void main(String[] args) {
    try {
      try {
        throw new ArithmeticException("Root Cause: Division by zero");
      } catch (ArithmeticException e) {
        throw new RuntimeException("Wrapped Exception", e);
      }
    } catch (RuntimeException ex) {
      System.out.println(ex.getMessage());
      System.out.println("Caused by: " + ex.getCause());
    }
  }
}
```

Output:

```
Wrapped Exception
Caused by: java.lang.ArithmeticException: Root Cause: Division by zero
```


3.13.4 Best Practices for Exception Handling

3.13.4.0.1 Best Practices:

- **Catch Specific Exceptions**: Always catch specific exceptions instead of using generic `Exception`.

```
try {
  int result = 10 / 0;
} catch (ArithmeticException e) {
  System.out.println("Handle division by zero");
}
```

- **Avoid Swallowing Exceptions**: Do not leave catch blocks empty. Log or handle exceptions appropriately.

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

- **Use Finally for Resource Cleanup**: Always close resources (e.g., files, streams) in the `finally` block or use try-with-resources.

```
try {
  // File operation
} finally {
  System.out.println("Cleaning up...");
}
```

- **Propagate Exceptions Properly**: Use `throws` in method declarations for unchecked exceptions.
- **Use Custom Exceptions for Clarity**: Create custom exceptions for business-specific issues.
- **Log Exceptions with Stack Trace**: Always log exceptions for debugging purposes.

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

3.13.4.0.2 Summary of Exception Handling and Error Management

- Java provides a robust mechanism to handle runtime errors using `try`, `catch`, `finally`, `throw`, and `throws`.
- Exceptions are categorized into checked exceptions, unchecked exceptions, and errors.

- Use custom exceptions and exception chaining to provide clarity and preserve the root cause of errors.
- Follow best practices to write clean, maintainable, and fault-tolerant code.

By understanding exception handling and implementing these principles effectively, developers can build reliable and resilient Java applications.

3.14 Generics in Java

Generics in Java provide a way to define classes, interfaces, and methods with type parameters. This allows developers to write reusable, type-safe code while avoiding the need for explicit type casting. Generics were introduced in Java 5 and are widely used in the Java Collections Framework.

3.14.1 Understanding Generics and Type Safety

3.14.1.0.1 What are Generics? Generics allow a class, interface, or method to operate on types. By using generics, you can define a type parameter T, which can be replaced with an actual type when the class or method is used.

3.14.1.0.2 Benefits of Generics:

- Type Safety: Errors are caught at compile time rather than runtime.
- Code Reusability: Write generic code that works with different data types.
- Elimination of Type Casting: No need to explicitly cast types when retrieving objects.

3.14.1.0.3 Example Without Generics (Legacy Code):

```
import java.util.ArrayList;

public class LegacyExample {
  public static void main(String[] args) {
    ArrayList list = new ArrayList();
    list.add("Hello");
    list.add(123); // Allowed without type safety

    // Requires casting
    String value = (String) list.get(0);
    System.out.println(value);
  }
}
```

3.14.1.0.4 Example With Generics: Generics eliminate the need for explicit type casting.

```
import java.util.ArrayList;

public class GenericsExample {
  public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<>();
```



```
    list.add("Hello");
    // list.add(123); // Compile-time error

    String value = list.get(0); // No casting required
    System.out.println(value);
  }
}
```

Output:

```
Hello
```

3.14.2 Bounded Type Parameters and Wildcards

3.14.2.0.1 Bounded Type Parameters Bounded type parameters restrict the types that can be passed to a generic type. Use the `extends` keyword to define an upper bound.

3.14.2.0.2 Example of Bounded Type Parameters:

```
class Box<T extends Number> { // T must be a subclass of Number
  private T value;

  public Box(T value) {
    this.value = value;
  }

  public void display() {
    System.out.println("Value: " + value);
  }
}

public class BoundedTypeExample {
  public static void main(String[] args) {
    Box<Integer> intBox = new Box<>(123);
    Box<Double> doubleBox = new Box<>(45.67);
    // Box<String> stringBox = new Box<>("Hello"); // Compile-time error

    intBox.display();
    doubleBox.display();
  }
}
```

Output:

```
Value: 123
Value: 45.67
```


3.14.2.0.3 Wildcards in Generics

Wildcards (`?`) allow you to use generics more flexibly. They can represent an unknown type.

- Upper Bounded Wildcard: `? extends Type` (accepts Type or its subclasses).
- Lower Bounded Wildcard: `? super Type` (accepts Type or its superclasses).
- Unbounded Wildcard: `?` (accepts any type).

3.14.2.0.4 Example of Wildcards:

```
import java.util.List;
import java.util.ArrayList;

class WildcardExample {
  // Upper Bounded Wildcard
  public static void displayNumbers(List<? extends Number> list) {
    for (Number n : list) {
      System.out.println(n);
    }
  }

  public static void main(String[] args) {
    List<Integer> intList = new ArrayList<>();
    intList.add(10);
    intList.add(20);

    List<Double> doubleList = new ArrayList<>();
    doubleList.add(1.5);
    doubleList.add(2.5);

    System.out.println("Integer List:");
    displayNumbers(intList); // Accepts Integer because it extends Number

    System.out.println("Double List:");
    displayNumbers(doubleList); // Accepts Double because it extends Number
  }
}
```

Output:

```
Integer List:
10
20
Double List:
1.5
2.5
```


3.14.3 Generics in Collections and Methods

3.14.3.0.1 Generics in Collections Generics are widely used in the Java Collections Framework to ensure type safety. Examples include `ArrayList<T>`, `HashMap<K, V>`, and `TreeSet<T>`.

3.14.3.0.2 Example: Generic Collections with Type Safety

```
import java.util.ArrayList;

public class GenericCollections {
  public static void main(String[] args) {
    ArrayList<String> names = new ArrayList<>();
    names.add("Alice");
    names.add("Bob");

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

Output:

```
Alice
Bob
```

3.14.3.0.3 Generic Methods You can define methods with generic type parameters to make them more versatile.

3.14.3.0.4 Example of a Generic Method:

```
class Utility {
  // Generic method
  public static <T> void printArray(T[] array) {
    for (T element : array) {
      System.out.print(element + " ");
    }
    System.out.println();
  }
}

public class GenericMethodExample {
  public static void main(String[] args) {
    Integer[] intArray = {1, 2, 3};
    String[] strArray = {"Hello", "World"};

    System.out.println("Integer Array:");
    Utility.printArray(intArray);

    System.out.println("String Array:");
```



```
    Utility.printArray(strArray);
  }
}
```

Output:

```
Integer Array:
1 2 3
String Array:
Hello World
```

3.14.4 Best Practices for Generics

3.14.4.0.1 Best Practices:

- Always use generics to ensure type safety and eliminate the need for type casting.
- Prefer `? extends Type` for read-only operations and `? super Type` for write operations when using wildcards.
- Use bounded type parameters for flexibility with constraints (e.g., `<T extends Number>`).
- Avoid using raw types (e.g., `ArrayList` instead of `ArrayList<T>`).
- Write generic methods for reusable utilities.

3.14.4.0.2 Summary of Generics in Java

- Generics provide type safety, reusability, and eliminate explicit type casting.
- Use bounded type parameters (`<T extends Type>`) to restrict the allowed types for generics.
- Wildcards (`?`, `? extends`, `? super`) allow flexibility in working with unknown types.
- Generics are extensively used in the Collections Framework and can be applied to methods for versatility.

By understanding generics and their features, developers can write clean, reusable, and type-safe code in Java.

3.15 Java API Documentation and javadoc

Java provides a powerful tool called javadoc to generate API documentation from source code comments. By writing structured documentation comments, developers can create comprehensive HTML documentation that improves code understanding and maintainability.

This chapter explains how to use javadoc, generate API documentation, and include custom javadoc tags with practical examples.

3.15.1 Generating API Documentation with javadoc

3.15.1.0.1 What is javadoc? javadoc is a command-line tool provided by the JDK to generate HTML-based documentation for Java classes, interfaces, and methods. It parses documentation comments (`/ ... */`) and produces human-readable API documentation.

3.15.1.0.2 Writing Documentation Comments Documentation comments are written using `/ ... */` syntax. Inside the comments, you can include descriptions, tags, and annotations to describe classes, methods, and fields.

3.15.1.0.3 Example: Writing javadoc Comments

```
package com.example.documentation;

/
* The {@code Calculator} class provides basic arithmetic operations.
* <p>
* This class includes methods for addition, subtraction, multiplication,
* and division of two numbers.
* </p>
*
* @author John Doe
* @version 1.0
* @since 2024-01-01
*/
public class Calculator {

  /
  * Adds two integers.
  *
  * @param a the first number
  * @param b the second number
  * @return the sum of {@code a} and {@code b}
  */
  public int add(int a, int b) {
    return a + b;
  }
```



```
  /
  * Subtracts one integer from another.
  *
  * @param a the first number
  * @param b the second number
  * @return the result of {@code a} minus {@code b}
  */
  public int subtract(int a, int b) {
    return a - b;
  }

  /
  * Divides two integers.
  *
  * @param a the numerator
  * @param b the denominator (must not be zero)
  * @return the result of {@code a} divided by {@code b}
  * @throws ArithmeticException if {@code b} is zero
  */
  public double divide(int a, int b) throws ArithmeticException {
    if (b == 0) {
      throw new ArithmeticException("Division by zero is not allowed.");
    }
    return (double) a / b;
  }

  /
  * Multiplies two integers.
  *
  * @param a the first number
  * @param b the second number
  * @return the product of {@code a} and {@code b}
  */
  public int multiply(int a, int b) {
    return a * b;
  }
}
```

3.15.1.0.4 Generating Documentation Using javadoc

To generate HTML documentation from the above class:

```
# Generate javadoc for the Calculator class
javadoc -d docs com/example/documentation/Calculator.java
```

- `-d docs`: Specifies the output directory for the generated HTML documentation.
- `Calculator.java`: The source file containing javadoc comments.

3.15.1.0.5 Viewing the Documentation

Open the generated HTML file (e.g., `docs/index.html`) in a web browser. The documentation includes:

- Class-level description.

- Method summaries, parameters, and return types.
- Exception details and additional metadata like `@author`, `@version`, and `@since`.

3.15.2 Custom javadoc Tags and Examples

3.15.2.0.1 Commonly Used javadoc Tags

- **@author**: Specifies the author of the class.
- **@version**: Specifies the version of the class or method.
- **@param**: Describes a method parameter.
- **@return**: Describes the method's return value.
- **@throws** or **@exception**: Describes an exception thrown by the method.
- **@see**: Links to another class, method, or URL.
- **@since**: Specifies the version when the feature was added.
- **@deprecated**: Marks a method or class as deprecated.
- **@code**: Displays text as code (e.g., inline code snippets).

3.15.2.0.2 Example: Custom Tags and Deprecated Methods

```
package com.example.documentation;

/
* Utility class for mathematical calculations.
*
* @author John Doe
* @version 1.1
* @since 2024-01-01
*/
public class MathUtils {

  /
  * Calculates the square of a number.
  *
  * @param number the input number
  * @return the square of the input number
  */
  public int square(int number) {
    return number * number;
  }
```



```
  /
  * Calculates the cube of a number.
  *
  * @param number the input number
  * @return the cube of the input number
  */
  public int cube(int number) {
    return number * number * number;
  }

  /
  * Calculates the square root of a number.
  *
  * @param number the input number
  * @return the square root of the input number
  * @deprecated Use {@link java.lang.Math#sqrt(double)} instead.
  */
  @Deprecated
  public double squareRoot(int number) {
    return Math.sqrt(number);
  }
}
```

3.15.2.0.3 Adding Custom Tags

Java supports custom javadoc tags using the `-tag` option.

```
javadoc -d docs -tag todo:a:"To Do:" com/example/documentation/MathUtils.java
```

Here, `@todo` becomes a custom tag that can be included in the documentation comments.

```
/
* @todo Optimize this method for better performance.
*/
```

3.15.3 Using Code Examples in javadoc

3.15.3.0.1 Inline Code and Code Blocks

- Use `@code` for inline code snippets.
- Use `<pre>` tags for multi-line code blocks.

3.15.3.0.2 Example: Adding Code Snippets

```
package com.example.documentation;

/
* Example class to demonstrate code documentation.
*/
```



```
public class Example {

  /
  * Prints a simple message.
  * <p>
  * Usage:
  * <pre>{@code
    * Example example = new Example();
    * example.printMessage();
    * }</pre>
  * </p>
  */
  public void printMessage() {
    System.out.println("Hello, this is an example method!");
  }
}
```

3.15.4 Best Practices for Writing javadoc

3.15.4.0.1 Best Practices

- Write clear and concise descriptions for classes, methods, and fields.
- Use standard tags (`@param`, `@return`, `@throws`) to document method details.
- Avoid redundant comments that restate the code.
- Use `@see` for references to related methods, classes, or external links.
- Document deprecated methods using `@deprecated` and suggest alternatives.
- Include code examples using `@code` and `<pre>` for better readability.

3.15.4.0.2 Summary of javadoc and Java API Documentation

- javadoc is a tool that generates HTML-based API documentation from source code.
- Use the `@param`, `@return`, `@throws`, and other tags to provide clear and structured documentation.
- Custom tags can be added using the `-tag` option for specialized documentation needs.
- Include inline code and examples with `@code` and `<pre>` for clarity.

- Good documentation improves code maintainability and helps other developers understand and use your code effectively.

By mastering javadoc and adopting best practices for documentation, developers can create well-documented, maintainable, and user-friendly Java APIs.

3.16 Working with Java Packages and Modules

Java provides two main mechanisms to organize and modularize code: Packages and the Java Platform Module System (JPMS). Packages allow grouping related classes, while JPMS, introduced in Java 9, offers better modularity for large applications.

This chapter explores how to organize code using packages and modules, with practical examples.

3.16.1 Organizing Code Using Packages

3.16.1.0.1 What is a Package? A package is a namespace that organizes related classes and interfaces. It helps:

- Avoid name conflicts.
- Group logically related components.
- Control access using access modifiers.

3.16.1.0.2 Defining a Package To create a package, use the `package` keyword at the top of a Java file.

```
// File: mypackage/MyClass.java
package mypackage;

public class MyClass {
  public void displayMessage() {
    System.out.println("Hello from MyClass in mypackage!");
  }
}
```

3.16.1.0.3 Using a Package To use a class from a package, use the `import` statement.

```
// File: Main.java
import mypackage.MyClass;

public class Main {
  public static void main(String[] args) {
    MyClass obj = new MyClass();
    obj.displayMessage();
  }
}
```


3.16.1.0.4 Directory Structure for Packages

In Java, the directory structure of a project should correspond to the package structure to ensure organization and compatibility with the Java compiler and runtime environment. Each package maps directly to a directory within the project's source folder.

For example:

- A class named `MyClass` within a package called `mypackage` should be placed inside a directory named `mypackage`. This directory must be located under the main source folder, commonly named `src`.
- A class that does not belong to any package, such as `Main`, should be placed directly in the `src` directory.

3.16.1.0.5 Compiling and Running a Package

Use the following commands to compile and run the package:

```
# Compile
javac -d . src/mypackage/MyClass.java src/Main.java

# Run
java Main
```

Output:

```
Hello from MyClass in mypackage!
```

3.16.2 Java Platform Module System (JPMS)

3.16.2.0.1 What is JPMS?

The Java Platform Module System (JPMS), introduced in Java 9, allows you to modularize your codebase. A module is a collection of packages, resources, and metadata defined in a `module-info.java` file.

3.16.2.0.2 Benefits of JPMS:

- Improved code organization and modularization.
- Strong encapsulation of internal classes.
- Reduced application size by including only required modules.
- Simplified dependency management.

3.16.3 Creating and Using Modules

3.16.3.0.1 Module Structure In Java, a module is a self-contained unit that encapsulates code and resources, improving modularity, dependency management, and access control. Each module has the following components:

- **Packages**: Grouped namespaces that organize related classes and interfaces.
- `module-info.java`: A descriptor file at the module's root, specifying the module's name, dependencies, and the packages it exports.

3.16.3.0.2 Example: Creating a Module **Step 1: Define the Module** To define a module, create a `module-info.java` file at the module's root. Use this file to declare the module's name and specify any packages that should be exported.

For example, consider a module named `myapp`. The `module-info.java` file would look like this:

```
// File: module-info.java
module myapp {
  exports mypackage; // Export the package to make it accessible to other modules
}
```

Within the module, include the necessary packages and classes. For example, create a package named `mypackage` and include a class `MyClass`:

```
// File: mypackage/MyClass.java
package mypackage;

public class MyClass {
  public void displayMessage() {
    System.out.println("Hello from MyClass in module myapp!");
  }
}
```

This defines a module named `myapp` that exports the package `mypackage`, making the `MyClass` class accessible to other modules.

Step 2: Use the Module To use the `myapp` module, create another module that consumes it. This consuming module will declare a dependency on `myapp` in its `module-info.java` file.

For example, consider a module named `appuser`:

```
// File: module-info.java
module appuser {
  requires myapp; // Declare dependency on module myapp
}
```


Within the `appuser` module, create a package named `userpackage` and add a `Main` class to use the functionality provided by `myapp`:

```
// File: userpackage/Main.java
package userpackage;

import mypackage.MyClass;

public class Main {
  public static void main(String[] args) {
    MyClass obj = new MyClass();
    obj.displayMessage();
  }
}
```

The `Main` class imports the `MyClass` from `mypackage`, which is part of the `myapp` module. When the program is executed, the output will be:

```
Hello from MyClass in module myapp!
```

3.16.3.0.3 Explanation of the Example

This example illustrates how to create and use Java modules:

- The `myapp` module defines and exports the `mypackage` package, containing the `MyClass` class.
- The `appuser` module declares a dependency on `myapp` and uses the `MyClass` functionality in its `Main` class.
- Explicit module declarations and package exports ensure clear dependencies and controlled access between modules.

3.16.3.0.4 Compiling and Running Modules Step 1: Compile Modules

```
# Compile module myapp
javac -d mods/myapp myapp/module-info.java myapp/mypackage/MyClass.java

# Compile module appuser
javac --module-path mods -d mods/appuser appuser/module-info.java appuser/userpackage/Main.java
```

Step 2: Run the Application

```
java --module-path mods -m appuser/userpackage.Main
```

Output:

```
Hello from MyClass in module myapp!
```


3.16.4 Encapsulation and Exports in Modules

3.16.4.0.1 Strong Encapsulation By default, packages in a module are not accessible to other modules. You must explicitly export packages using the `exports` keyword in `module-info.java`.

```
// Exports a package
module myapp {
  exports mypackage;
}
```

3.16.4.0.2 Using `requires` A module specifies its dependencies using the `requires` keyword.

```
// File: module-info.java
module appuser {
  requires myapp; // Dependency on myapp module
}
```

3.16.4.0.3 Restricting Access with `opens` The `opens` keyword allows runtime reflection access to a package but does not export it for compilation.

3.16.5 Best Practices for Packages and Modules

3.16.5.0.1 Best Practices for Packages:

- Use meaningful and hierarchical package names (e.g., `com.company.module`).
- Keep related classes and interfaces in the same package.
- Use access modifiers (`public`, `private`) to restrict access to package members.

3.16.5.0.2 Best Practices for Modules:

- Start with modularization for larger projects with multiple dependencies.
- Export only necessary packages using the `exports` directive.
- Use `requires` to specify module dependencies explicitly.
- Use `opens` for reflection-based frameworks (e.g., serialization).

3.16.5.0.3 Summary of Packages and Modules

- Packages organize classes into namespaces, improving code structure and avoiding name conflicts.
- The Java Platform Module System (JPMS), introduced in Java 9, modularizes applications, enabling better encapsulation and dependency management.
- Modules use `module-info.java` to define dependencies and exported packages.
- Proper use of packages and modules enhances code maintainability, scalability, and readability.

By mastering packages and JPMS, developers can create well-organized and modularized Java applications, improving both code management and runtime efficiency.

Chapter 4

Java Collections and Maps

4.1 Introduction to Collections Framework

The Java Collections Framework is a unified architecture for storing, manipulating, and processing groups of objects. It provides classes and interfaces that make it easier to manage dynamic collections of data, such as lists, sets, and maps. The framework is part of the `java.util` package.

4.1.1 Overview of Collections API

4.1.1.0.1 What is the Collections Framework? The Collections Framework in Java is a set of classes and interfaces that provide efficient ways to manage collections (groups of objects). It includes:

- Interfaces: Define abstract data types (e.g., `List`, `Set`, `Map`).
- Classes: Implement these interfaces (e.g., `ArrayList`, `HashSet`, `HashMap`).
- Algorithms: Utility methods for sorting, searching, and manipulating collections (e.g., `Collections.sort()`).

4.1.1.0.2 Key Interfaces in the Collections Framework The core interfaces of the Collections Framework are:

- **`Collection`**: Root interface for all collection types.
- **`List`**: An ordered collection that allows duplicates (e.g., `ArrayList`, `LinkedList`).
- **`Set`**: A collection that does not allow duplicate elements (e.g., `HashSet`, `TreeSet`).
- **`Map`**: A collection that stores key-value pairs (e.g., `HashMap`, `TreeMap`).
- **`Queue`**: A collection that follows FIFO (First-In-First-Out) principles (e.g., `LinkedList`, `PriorityQueue`).

4.1.1.0.3 Hierarchy of the Collections Framework The Java Collections Framework provides a unified architecture for managing and manipulating groups of objects. It includes interfaces, classes, and algorithms for data structures, enabling developers to handle collections efficiently. Below is an enhanced and detailed description of the hierarchy for collections, focusing on the key types: **List**, **Set**, **Queue**, and **Map**.

The `Collection` interface is the root interface of the collections hierarchy, providing the foundation for the **List**, **Set**, and **Queue** subinterfaces. Each subinterface represents a distinct type of collection with its own characteristics and behaviors.

The `List` interface represents an ordered collection that allows duplicate elements. It is implemented by the following classes:

- **ArrayList**: A resizable array implementation that offers fast random access to elements. It is best suited for scenarios where frequent reads are required but writes or removals are less frequent.
- **LinkedList**: A doubly-linked list implementation that provides efficient insertions and deletions at the cost of slower random access. It is ideal for use cases requiring frequent updates to the list structure.
- **Vector**: A synchronized, thread-safe implementation of a dynamic array. Though rarely used today, it is suitable for legacy applications requiring thread safety.

The `Set` interface represents a collection that does not allow duplicate elements. Implementations include:

- **HashSet**: A hash table-based implementation that offers constant-time performance for basic operations like add, remove, and contains. The order of elements is not guaranteed.
- **LinkedHashSet**: Extends `HashSet` and maintains a linked list of elements, preserving the insertion order.
- **TreeSet**: A navigable set backed by a `TreeMap`. It guarantees that elements are sorted in their natural order or according to a provided comparator.

The `Queue` interface represents a collection designed for holding elements prior to processing, typically in a FIFO (First-In-First-Out) order. Key implementations include:

- **PriorityQueue**: A priority heap-based implementation where elements are ordered according to their natural order or by a comparator. It is commonly used in scheduling and resource management.
- **LinkedList**: Implements both the `List` and `Queue` interfaces. It can function as a deque (double-ended queue), providing flexibility for insertion and deletion at both ends.

The `Map` interface represents a collection of key-value pairs. Unlike `Collection`, `Map` forms a separate hierarchy, and its implementations include:

- **HashMap**: A hash table-based implementation that provides constant-time performance for basic operations. The order of keys and values is not guaranteed.
- **LinkedHashMap**: Extends `HashMap` and maintains a linked list of entries, preserving the insertion order or access order (if configured).
- **TreeMap**: A red-black tree-based implementation that orders keys in their natural order or according to a specified comparator. It supports navigational methods like `headMap`, `tailMap`, and `subMap`.

4.1.2 Differences Between List, Set, and Map

4.1.2.0.1 Comparison of List, Set, and Map

Feature	List	Set	Map
Duplicates	Allows duplicates	No duplicates	Keys: Unique, Values:
Order	Preserves insertion order	No guaranteed order	No guaranteed order
Null Values	Allows null values	Allows one null element	Allows one null key
Implementation	`ArrayList`, `LinkedList`	`HashSet`, `TreeSet`	`HashMap`, `TreeMap`

4.1.2.0.2 Example of List (ArrayList) A `List` preserves the order of elements and allows duplicates.

```
import java.util.ArrayList;

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

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

Output:

```
List: [Apple, Banana, Apple]
```


4.1.2.0.3 Example of Set (HashSet)

A `Set` does not allow duplicates and does not guarantee order.

```
import java.util.HashSet;

public class SetExample {
  public static void main(String[] args) {
    HashSet<String> set = new HashSet<>();
    set.add("Apple");
    set.add("Banana");
    set.add("Apple"); // Duplicate ignored

    System.out.println("Set: " + set);
  }
}
```

Output:

```
Set: [Apple, Banana]
```

4.1.2.0.4 Example of Map (HashMap)

A `Map` stores key-value pairs.

```
import java.util.HashMap;

public class MapExample {
  public static void main(String[] args) {
    HashMap<Integer, String> map = new HashMap<>();
    map.put(1, "Apple");
    map.put(2, "Banana");
    map.put(1, "Cherry"); // Replaces previous value for key 1

    System.out.println("Map: " + map);
  }
}
```

Output:

```
Map: {1=Cherry, 2=Banana}
```

4.1.3 Iterators and the `Iterable` Interface

4.1.3.0.1 What is an Iterator?

An `Iterator` is an object used to traverse elements in a collection. It provides methods to:

- Retrieve elements sequentially.
- Remove elements during iteration.

4.1.3.0.2 Methods of the Iterator Interface

- `hasNext()`: Returns `true` if there are more elements.
- `next()`: Returns the next element.
- `remove()`: Removes the current element.

4.1.3.0.3 Example of Using an Iterator

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

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

    // Using Iterator
    Iterator<String> iterator = list.iterator();
    while (iterator.hasNext()) {
      String fruit = iterator.next();
      System.out.println(fruit);
      if (fruit.equals("Banana")) {
        iterator.remove(); // Remove element
      }
    }

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

Output:

```
Apple
Banana
Cherry
Updated List: [Apple, Cherry]
```

4.1.3.0.4 The `Iterable` Interface

The `Iterable` interface allows a collection to be iterated using an enhanced `for` loop.

```
import java.util.ArrayList;

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



```
    list.add("Banana");
    list.add("Cherry");

    for (String fruit : list) { // Enhanced for loop using Iterable
      System.out.println(fruit);
    }
  }
}
```

Output:

```
Apple
Banana
Cherry
```

4.1.4 Best Practices for Collections Framework

4.1.4.0.1 Best Practices:

- Use the appropriate collection type based on requirements (e.g., `List` for ordered elements, `Set` for unique elements).
- Prefer `ArrayList` for faster random access and `LinkedList` for frequent insertions/deletions.
- Always iterate over collections using `Iterator` or enhanced for-loops for safety.
- Use `Map` for key-value pairs, but choose the correct implementation (`HashMap` for unsorted keys, `TreeMap` for sorted keys).
- Avoid using raw types; always use generics to ensure type safety.

```
ArrayList<String> list = new ArrayList<>();
```

4.1.4.0.2 Summary of the Collections Framework

- The Java Collections Framework provides interfaces (`List`, `Set`, `Map`) and their implementations for managing groups of objects.
- `List` allows duplicates and preserves order, `Set` ensures uniqueness, and `Map` works with key-value pairs.
- Iterators allow safe and sequential traversal of collections.
- Use the enhanced `for` loop for cleaner iteration.

By mastering the Collections Framework, developers can efficiently store, manipulate, and process data in Java programs.

4.2 Lists: ArrayList and LinkedList

In the Java Collections Framework, `List` is an ordered collection that allows duplicates and provides positional access to elements. The two most commonly used implementations of the `List` interface are `ArrayList` and `LinkedList`. This chapter explains their features, applications, differences, and efficient iteration techniques.

4.2.1 Features and Applications of ArrayList

4.2.1.0.1 What is an `ArrayList`?

`ArrayList` is a resizable array implementation of the `List` interface. Unlike arrays, it can grow dynamically as elements are added.

4.2.1.0.2 Features of `ArrayList`:

- Allows random access to elements using indices.
- Allows duplicate values and preserves insertion order.
- Provides dynamic resizing (managed internally).
- Performance for retrieval is fast ($O(1)$) but slower for insertions/deletions in the middle ($O(n)$).

4.2.1.0.3 Example: Creating and Using an `ArrayList`

```
import java.util.ArrayList;

public class ArrayListExample {
  public static void main(String[] args) {
    // Create an ArrayList of Strings
    ArrayList<String> fruits = new ArrayList<>();
    fruits.add("Apple");
    fruits.add("Banana");
    fruits.add("Cherry");

    // Insert a new element
    fruits.add(1, "Orange"); // Insert at index 1

    // Access elements
    System.out.println("Element at index 2: " + fruits.get(2));

    // Iterate using for-each loop
    for (String fruit : fruits) {
      System.out.println(fruit);
    }

    // Remove an element
    fruits.remove("Banana");
```



```
    System.out.println("After removal: " + fruits);
  }
}
```

Output:

```
Element at index 2: Cherry
Apple
Orange
Banana
Cherry
After removal: [Apple, Orange, Cherry]
```

4.2.1.0.4 Applications of `ArrayList`:

- Best suited for scenarios where random access to elements is needed.
- Ideal for storing a fixed number of elements with frequent retrievals.
- Used in dynamic arrays where resizing is necessary.

4.2.2 When to Use LinkedList vs ArrayList

4.2.2.0.1 What is a `LinkedList`? `LinkedList` is a doubly-linked list implementation of the `List` and `Deque` interfaces. It provides better performance for frequent insertions and deletions.

4.2.2.0.2 Key Features of `LinkedList`:

- Elements are stored as nodes, with each node pointing to the next and previous nodes.
- Insertions and deletions are efficient ($O(1)$ for the middle), but access time is slower ($O(n)$).
- Supports both list and queue operations.

4.2.2.0.3 Example: Creating and Using a `LinkedList`

```
import java.util.LinkedList;

public class LinkedListExample {
  public static void main(String[] args) {
    // Create a LinkedList of Strings
    LinkedList<String> cities = new LinkedList<>();
    cities.add("New York");
    cities.add("Los Angeles");
    cities.add("Chicago");

    // Add elements at specific positions
    cities.addFirst("San Francisco");
    cities.addLast("Houston");

    // Access elements
    System.out.println("First City: " + cities.getFirst());
    System.out.println("Last City: " + cities.getLast());

    // Iterate using for-each loop
    for (String city : cities) {
      System.out.println(city);
    }

    // Remove elements
    cities.remove("Los Angeles");
    System.out.println("After removal: " + cities);
  }
}
```

Output:

```
First City: San Francisco
Last City: Houston
San Francisco
New York
Los Angeles
Chicago
Houston
After removal: [San Francisco, New York, Chicago, Houston]
```

4.2.2.0.4 When to Use `ArrayList` vs `LinkedList`:

Criteria	**ArrayList**	**LinkedList**
Data Storage	Dynamic array	Doubly-linked list
Access Time	Fast ($O(1)$)	Slow ($O(n)$)
Insertion/Deletion	Slow in the middle ($O(n)$)	Fast anywhere ($O(1)$)
Memory Overhead	Less memory overhead	More memory overhead (node pointers)
Best Use Case	Frequent access and retrieval	Frequent insertions and deletions

4.2.3 Iterating Lists Using Streams and Iterators

4.2.3.0.1 Iterating Using an Iterator The `Iterator` interface provides a way to iterate through elements of a list safely.

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

public class IteratorExample {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<>();
    numbers.add(1);
    numbers.add(2);
    numbers.add(3);

    Iterator<Integer> iterator = numbers.iterator();
    while (iterator.hasNext()) {
      int number = iterator.next();
      System.out.println(number);
      if (number == 2) {
        iterator.remove(); // Safe removal during iteration
      }
    }

    System.out.println("After removal: " + numbers);
  }
}
```

Output:

```
1
2
3
After removal: [1, 3]
```

4.2.3.0.2 Iterating Using Java Streams (Java 8+) Java Streams simplify iteration and support functional programming.

```
import java.util.ArrayList;

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

    // Stream to filter and display elements
    fruits.stream()
    .filter(fruit -> fruit.startsWith("A"))
    .forEach(System.out::println);
  }
}
```


Output:

```
Apple
```

4.2.4 Best Practices for Using Lists

4.2.4.0.1 Best Practices:

- Use `ArrayList` when frequent random access is required.
- Use `LinkedList` when frequent insertions and deletions are expected.
- Prefer `Iterator` for safe removal during iteration.
- Use Streams for concise and functional-style list processing.
- Avoid using raw types; always use generics to ensure type safety.

4.2.4.0.2 Avoiding Concurrent Modification Exceptions: When modifying a list during iteration, always use an `Iterator` or concurrent collection classes.

4.2.4.0.3 Summary of Lists: ArrayList and LinkedList

- `ArrayList`: Provides fast random access but slower insertions/deletions.
- `LinkedList`: Provides efficient insertions and deletions but slower access time.
- Use `Iterator` for safe traversal and modification.
- Use Java Streams to simplify list operations with modern functional programming techniques.

By understanding the features, use cases, and iteration techniques of `ArrayList` and `LinkedList`, developers can choose the appropriate implementation for their specific requirements.

4.3 Sets: HashSet and TreeSet

In the Java Collections Framework, a Set is a collection that does not allow duplicate elements. It is useful when you need to store unique elements and perform operations like searching, insertion, and deletion efficiently. The two most commonly used implementations of the `Set` interface are `HashSet` and `TreeSet`.

4.3.1 Unique Element Storage in Sets

4.3.1.0.1 What is a Set? A `Set` is an unordered collection of unique elements. Java does not allow duplicate elements in a `Set`.

4.3.1.0.2 Key Characteristics of Sets:

- No duplicate elements are allowed.
- `HashSet` does not guarantee any specific order.
- `TreeSet` maintains elements in sorted order.

4.3.1.0.3 Example of Unique Element Storage Using `HashSet`:

```
import java.util.HashSet;

public class HashSetExample {
  public static void main(String[] args) {
    // Create a HashSet of Strings
    HashSet<String> set = new HashSet<>();
    set.add("Apple");
    set.add("Banana");
    set.add("Cherry");
    set.add("Apple"); // Duplicate element, ignored

    System.out.println("HashSet Elements: " + set);
  }
}
```

Output:

```
HashSet Elements: [Apple, Banana, Cherry]
```

4.3.1.0.4 Explanation: In the `HashSet`, duplicates are ignored, and elements are stored in an unordered fashion.

4.3.2 TreeSet and SortedSet for Ordered Data

4.3.2.0.1 What is a `TreeSet`? `TreeSet` is an implementation of the `SortedSet` interface. It stores elements in sorted (natural) order or according to a custom comparator.

4.3.2.0.2 Key Features of `TreeSet`:

- Elements are sorted in ascending (natural) order by default.
- Does not allow duplicates.
- Provides efficient search, insertion, and deletion ($O(\log n)$).

4.3.2.0.3 Example of `TreeSet` for Ordered Data:

```
import java.util.TreeSet;

public class TreeSetExample {
  public static void main(String[] args) {
    // Create a TreeSet of Integers
    TreeSet<Integer> treeSet = new TreeSet<>();
    treeSet.add(5);
    treeSet.add(1);
    treeSet.add(3);
    treeSet.add(2);
    treeSet.add(4);

    System.out.println("TreeSet Elements (Sorted): " + treeSet);
  }
}
```

Output:

```
TreeSet Elements (Sorted): [1, 2, 3, 4, 5]
```

4.3.2.0.4 Explanation: `TreeSet` automatically sorts the elements in their natural order (ascending for integers).

4.3.3 Custom Comparators for `TreeSet`

4.3.3.0.1 What is a Comparator? A Comparator is used to define custom sorting logic for a collection. When using `TreeSet`, you can pass a custom comparator to sort elements based on specific criteria.

4.3.3.0.2 Implementing Custom Sorting with `TreeSet`:

```
import java.util.Comparator;
import java.util.TreeSet;

class Person {
  String name;
  int age;

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

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

public class TreeSetCustomComparator {
  public static void main(String[] args) {
    // Custom comparator to sort by age
    Comparator<Person> ageComparator = new Comparator<>() {
      @Override
      public int compare(Person p1, Person p2) {
        return Integer.compare(p1.age, p2.age);
      }
    };

    TreeSet<Person> people = new TreeSet<>(ageComparator);
    people.add(new Person("Alice", 30));
    people.add(new Person("Bob", 25));
    people.add(new Person("Charlie", 35));

    System.out.println("People sorted by age:");
    for (Person p : people) {
      System.out.println(p);
    }
  }
}
```

Output:

```
People sorted by age:
Bob (25)
Alice (30)
Charlie (35)
```

4.3.3.0.3 Explanation:

In this example:

- A custom `Comparator` is provided to sort `Person` objects by age.
- The `TreeSet` ensures elements are sorted according to the comparator logic.

4.3.4 Differences Between `HashSet` and `TreeSet`

Feature	`HashSet`	`TreeSet`
Ordering	No guaranteed order	Elements are sorted (natural or
Performance	Faster for add, remove, and search ($O(1)$)	Slower due to sorting ($O(\log n)$)
Null Values	Allows a single null element	Does not allow null elements
Implementation	Backed by a hash table	Backed by a red-black tree
Best Use Case	When order does not matter	When sorted order is required

4.3.5 Iterating Through a Set Using Streams and Iterators

4.3.5.0.1 Iterating Using an Iterator:

```
import java.util.HashSet;
import java.util.Iterator;

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

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

4.3.5.0.2 Iterating Using Streams (Java 8+): Streams simplify iteration by providing functional programming constructs.

```
import java.util.TreeSet;

public class StreamIterationExample {
  public static void main(String[] args) {
    TreeSet<Integer> numbers = new TreeSet<>();
    numbers.add(1);
    numbers.add(2);
    numbers.add(3);

    // Using Stream API
    numbers.stream().forEach(System.out::println);
  }
}
```

Output:


```
Apple
Banana
Cherry

1
2
3
```

4.3.6 Best Practices for Using Sets

4.3.6.0.1 Best Practices:

- Use `HashSet` when ordering is not important and performance is critical.
- Use `TreeSet` when a sorted order is required.
- Provide a proper implementation of `hashCode()` and `equals()` when working with custom objects in a `HashSet`.
- Use a custom `Comparator` for sorting in `TreeSet`.
- Avoid storing null elements in `TreeSet`, as it does not allow them.

4.3.6.0.2 Summary of Sets: HashSet and TreeSet

- `HashSet`: Stores unique elements in an unordered manner and is efficient for fast operations.
- `TreeSet`: Stores unique elements in a sorted order, using natural or custom sorting.
- Use iterators or Java Streams to traverse elements in a `Set`.
- Choose `HashSet` for performance and `TreeSet` for sorting requirements.

By understanding and using `HashSet` and `TreeSet` effectively, developers can handle collections of unique elements efficiently and in a way that best suits their application's requirements.

4.4 Maps: HashMap and TreeMap

In the Java Collections Framework, a Map is a collection that stores key-value pairs. Each key maps to exactly one value, and duplicate keys are not allowed. The two most commonly used implementations of the `Map` interface are `HashMap` and `TreeMap`.

4.4.1 Storing Key-Value Pairs

4.4.1.0.1 What is a Map? A `Map` is a data structure that allows storing, retrieving, and managing key-value pairs. Keys must be unique, but values can be duplicated.

4.4.1.0.2 Key Features of a Map:

- A key maps to a single value.
- Duplicate keys are not allowed; inserting a new value with an existing key replaces the old value.
- `HashMap` is unordered, while `TreeMap` sorts keys in natural or custom order.

4.4.1.0.3 Example of Storing Key-Value Pairs Using `HashMap`:

```
import java.util.HashMap;

public class HashMapExample {
  public static void main(String[] args) {
    // Create a HashMap to store key-value pairs
    HashMap<Integer, String> map = new HashMap<>();

    // Add key-value pairs
    map.put(1, "Apple");
    map.put(2, "Banana");
    map.put(3, "Cherry");

    // Retrieve a value by key
    System.out.println("Key 2: " + map.get(2));

    // Iterate through the map
    for (Integer key : map.keySet()) {
      System.out.println("Key: " + key + ", Value: " + map.get(key));
    }
  }
}
```

Output:


```
Key 2: Banana
Key: 1, Value: Apple
Key: 2, Value: Banana
Key: 3, Value: Cherry
```

4.4.1.0.4 Explanation:

- The `HashMap` stores key-value pairs.
- The method `put()` adds new entries, and `get()` retrieves a value by its key.
- The order of entries is not guaranteed in a `HashMap`.

4.4.2 TreeMap for Sorted Data

4.4.2.0.1 What is a `TreeMap`? `TreeMap` is an implementation of the `Map` interface that stores keys in sorted order. It uses a Red-Black Tree to maintain the natural order of the keys or a custom order defined by a `Comparator`.

4.4.2.0.2 Key Features of `TreeMap`:

- Keys are stored in natural order (ascending) by default.
- Allows custom sorting using a `Comparator`.
- Does not allow `null` keys.

4.4.2.0.3 Example of `TreeMap` with Natural Sorting:

```
import java.util.TreeMap;

public class TreeMapExample {
  public static void main(String[] args) {
    // Create a TreeMap to store key-value pairs
    TreeMap<Integer, String> treeMap = new TreeMap<>();

    // Add key-value pairs
    treeMap.put(3, "Cherry");
    treeMap.put(1, "Apple");
    treeMap.put(2, "Banana");

    // Iterate through the map (keys are sorted)
    for (Integer key : treeMap.keySet()) {
      System.out.println("Key: " + key + ", Value: " + treeMap.get(key));
    }
  }
}
```


Output:

```
Key: 1, Value: Apple
Key: 2, Value: Banana
Key: 3, Value: Cherry
```

4.4.2.0.4 Using a Custom Comparator with `TreeMap`

You can define a custom sorting order for keys using a `Comparator`.

```
import java.util.TreeMap;
import java.util.Comparator;

public class CustomTreeMapExample {
  public static void main(String[] args) {
    // TreeMap with custom comparator for descending order
    TreeMap<Integer, String> treeMap = new TreeMap<>(Comparator.reverseOrder());

    treeMap.put(3, "Cherry");
    treeMap.put(1, "Apple");
    treeMap.put(2, "Banana");

    for (Integer key : treeMap.keySet()) {
      System.out.println("Key: " + key + ", Value: " + treeMap.get(key));
    }
  }
}
```

Output:

```
Key: 3, Value: Cherry
Key: 2, Value: Banana
Key: 1, Value: Apple
```

4.4.3 Advanced Map Features: `computeIfAbsent` and Merging

4.4.3.0.1 Using `computeIfAbsent`

The `computeIfAbsent` method computes a value for a given key if it is not already present in the map. It simplifies conditional logic for adding elements.

```
import java.util.HashMap;

public class ComputeIfAbsentExample {
  public static void main(String[] args) {
    HashMap<String, Integer> map = new HashMap<>();

    // Use computeIfAbsent to provide a default value for a key
    map.computeIfAbsent("Apple", key -> key.length());
    map.computeIfAbsent("Banana", key -> key.length());
```



```
    System.out.println("Map: " + map);
  }
}
```

Output:

```
Map: {Apple=5, Banana=6}
```

4.4.3.0.2 Using `merge`

The `merge` method combines values associated with a key using a given function.

```
import java.util.HashMap;

public class MergeExample {
  public static void main(String[] args) {
    HashMap<String, Integer> map = new HashMap<>();
    map.put("Apple", 3);
    map.put("Banana", 2);

    // Merge values
    map.merge("Apple", 2, (oldVal, newVal) -> oldVal + newVal);
    map.merge("Cherry", 5, (oldVal, newVal) -> oldVal + newVal);

    System.out.println("Map: " + map);
  }
}
```

Output:

```
Map: {Apple=5, Banana=2, Cherry=5}
```

4.4.3.0.3 Explanation:

- `computeIfAbsent`: Inserts a value if the key is absent.
- `merge`: Updates the value of a key based on a function or inserts it if the key is absent.

4.4.4 Best Practices for Using Maps

4.4.4.0.1 Best Practices:

- Use `HashMap` when order of keys is not important and performance is critical.
- Use `TreeMap` when keys need to be sorted naturally or using a custom comparator.
- Avoid using `null` keys in `TreeMap`.

- Use `computeIfAbsent` and `merge` for cleaner code when conditionally inserting or updating entries.
- Always override `hashCode()` and `equals()` when using custom objects as keys.

4.4.4.0.2 Summary of Maps: HashMap and TreeMap

- `HashMap`: Provides fast insertion, retrieval, and deletion but does not guarantee order.
- `TreeMap`: Maintains keys in sorted order (natural or custom) using a Red-Black Tree.
- Use advanced methods like `computeIfAbsent` and `merge` for dynamic updates.
- Choose the appropriate map implementation based on requirements for performance and ordering.

By mastering `HashMap`, `TreeMap`, and advanced map features, developers can efficiently manage and manipulate key-value pairs in Java applications.

4.5 Creating a Binary Tree and Implementing Traversals

Binary trees are fundamental data structures in computer science, consisting of nodes where each node has at most two children: a left child and a right child. This chapter covers the creation of a binary tree from scratch and explains different tree traversal techniques such as Breadth-First Search (BFS), Depth-First Search (DFS), and an introduction to algorithms like Dijkstra's Algorithm for graph-like structures.

4.5.1 Creating a Binary Tree from Scratch

4.5.1.0.1 Structure of a Binary Tree Node A binary tree consists of nodes where each node contains:

- **Data**: Value stored in the node.
- **Left Child**: Pointer to the left subtree.
- **Right Child**: Pointer to the right subtree.

4.5.1.0.2 Implementation of a Binary Tree Below is the Java implementation for creating a simple binary tree:

```
class Node {
  int data;
  Node left, right;

  public Node(int data) {
    this.data = data;
    this.left = this.right = null;
  }
}

class BinaryTree {
  Node root;

  // Constructor to initialize the binary tree
  public BinaryTree() {
    root = null;
  }

  // Add a simple display method to show the root node
  public void displayRoot() {
    if (root != null) {
      System.out.println("Root Node: " + root.data);
    } else {
      System.out.println("The tree is empty.");
    }
```



```
  }
}

public class BinaryTreeExample {
  public static void main(String[] args) {
    BinaryTree tree = new BinaryTree();

    // Manually constructing the binary tree
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);

    // Display the root node
    tree.displayRoot();
  }
}
```

4.5.1.0.3 Tree Structure:

The manually constructed binary tree looks like this:

```
    1
   / \
  2   3
 / \
4   5
```

Output:

```
Root Node: 1
```

4.5.2 Breadth-First Search (BFS) Traversal

4.5.2.0.1 What is BFS?

BFS (Breadth-First Search) explores nodes level by level from top to bottom and left to right. It uses a queue data structure to process nodes in the order they are discovered.

4.5.2.0.2 BFS Implementation:

```
import java.util.LinkedList;
import java.util.Queue;

class BFSBinaryTree {
  Node root;

  // BFS Traversal Method
  void bfs() {
    if (root == null) return;
```



```
    Queue<Node> queue = new LinkedList<>();
    queue.add(root);

    while (!queue.isEmpty()) {
      Node current = queue.poll();
      System.out.print(current.data + " ");

      if (current.left != null) {
        queue.add(current.left);
      }
      if (current.right != null) {
        queue.add(current.right);
      }
    }
  }
}

public class BFSExample {
  public static void main(String[] args) {
    BFSBinaryTree tree = new BFSBinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);

    System.out.println("BFS Traversal:");
    tree.bfs();
  }
}
```

Output:

```
BFS Traversal:
1 2 3 4 5
```

4.5.3 Depth-First Search (DFS) Traversals

4.5.3.0.1 What is DFS? DFS explores as far down a branch as possible before backtracking. DFS includes three traversal types:

- **Preorder (Root, Left, Right)**.
- **Inorder (Left, Root, Right)**.
- **Postorder (Left, Right, Root)**.

4.5.3.0.2 Implementing DFS Traversals:


```java
class DFSBinaryTree {
  Node root;

  // Preorder Traversal
  void preorder(Node node) {
    if (node == null) return;
    System.out.print(node.data + " ");
    preorder(node.left);
    preorder(node.right);
  }

  // Inorder Traversal
  void inorder(Node node) {
    if (node == null) return;
    inorder(node.left);
    System.out.print(node.data + " ");
    inorder(node.right);
  }

  // Postorder Traversal
  void postorder(Node node) {
    if (node == null) return;
    postorder(node.left);
    postorder(node.right);
    System.out.print(node.data + " ");
  }
}

public class DFSExample {
  public static void main(String[] args) {
    DFSBinaryTree tree = new DFSBinaryTree();
    tree.root = new Node(1);
    tree.root.left = new Node(2);
    tree.root.right = new Node(3);
    tree.root.left.left = new Node(4);
    tree.root.left.right = new Node(5);

    System.out.println("Preorder Traversal:");
    tree.preorder(tree.root);
    System.out.println("\nInorder Traversal:");
    tree.inorder(tree.root);
    System.out.println("\nPostorder Traversal:");
    tree.postorder(tree.root);
  }
}
```

Output:

```
Preorder Traversal:
1 2 4 5 3
Inorder Traversal:
4 2 5 1 3
Postorder Traversal:
4 5 2 3 1
```


4.5.4 Dijkstra's Algorithm for Graph-Like Trees

4.5.4.0.1 Dijkstra's Algorithm: Introduction Dijkstra's algorithm is typically used to find the shortest path in a graph. If the binary tree is treated as a weighted graph (edges have weights), Dijkstra's algorithm can be applied.

4.5.4.0.2 Implementing Dijkstra's Algorithm: Here is a conceptual implementation for weighted graphs:

```
import java.util.*;

class GraphNode {
  int vertex, weight;

  GraphNode(int vertex, int weight) {
    this.vertex = vertex;
    this.weight = weight;
  }
}

class Dijkstra {
  public void shortestPath(Map<Integer, List<GraphNode>> graph, int source) {
    PriorityQueue<GraphNode> pq = new PriorityQueue<>(Comparator.comparingInt(node -> node.weight));
    Map<Integer, Integer> distances = new HashMap<>();
    for (int key : graph.keySet()) distances.put(key, Integer.MAX_VALUE);
    distances.put(source, 0);

    pq.add(new GraphNode(source, 0));

    while (!pq.isEmpty()) {
      GraphNode current = pq.poll();

      for (GraphNode neighbor : graph.get(current.vertex)) {
        int newDist = distances.get(current.vertex) + neighbor.weight;
        if (newDist < distances.get(neighbor.vertex)) {
          distances.put(neighbor.vertex, newDist);
          pq.add(new GraphNode(neighbor.vertex, newDist));
        }
      }
    }

    System.out.println("Shortest distances: " + distances);
  }
}

public class DijkstraExample {
  public static void main(String[] args) {
    Map<Integer, List<GraphNode>> graph = new HashMap<>();
    graph.put(0, Arrays.asList(new GraphNode(1, 4), new GraphNode(2, 1)));
    graph.put(1, Arrays.asList(new GraphNode(3, 1)));
    graph.put(2, Arrays.asList(new GraphNode(1, 2), new GraphNode(3, 5)));
    graph.put(3, new ArrayList<>());

    Dijkstra dijkstra = new Dijkstra();
    dijkstra.shortestPath(graph, 0);
```



```
  }
}
```

Output:

```
Shortest distances: {0=0, 1=3, 2=1, 3=4}
```

4.5.5 Summary of Binary Trees and Traversals

- A binary tree can be implemented using nodes that have left and right child references.
- Traversals:
  - BFS uses a queue to explore the tree level by level.
  - DFS includes preorder, inorder, and postorder traversals.
- Dijkstra's algorithm can be applied to weighted graph-like trees to compute shortest paths.
- Choose BFS for level-order traversal and DFS for depth-based exploration.

By mastering these concepts and implementations, developers can efficiently manipulate and traverse binary trees and graphs in Java.

4.6 Sorting Algorithms, Big O Notation, and Running Time

Sorting algorithms are fundamental in computer science and are used to arrange data in a specific order (e.g., ascending or descending). Understanding the performance of these algorithms using Big O Notation is essential for choosing the right algorithm based on input size and requirements.

This chapter explains the concept of running time, Big O notation, and provides Java implementations for common sorting algorithms such as Bubble Sort, Merge Sort, Quick Sort, and Radix Sort.

4.6.1 Big O Notation and Running Time

4.6.1.0.1 What is Big O Notation? Big O Notation describes the upper bound of an algorithm's running time. It is used to measure the performance of an algorithm based on input size n. It focuses on the worst-case scenario.

4.6.1.0.2 Common Big O Complexities:

- $O(1)$: Constant time (e.g., accessing an array element).
- $O(\log n)$: Logarithmic time (e.g., binary search).
- $O(n)$: Linear time (e.g., iterating through an array).
- $O(n \log n)$: Log-linear time (e.g., merge sort, quick sort).
- $O(n^2)$: Quadratic time (e.g., bubble sort, selection sort).
- $O(n!)$: Factorial time (e.g., solving the traveling salesman problem).

4.6.2 Bubble Sort: Implementation and Analysis

4.6.2.0.1 What is Bubble Sort? Bubble Sort is a simple comparison-based algorithm where adjacent elements are compared, and larger values "bubble" to the end. It continues until the array is sorted.

4.6.2.0.2 Time Complexity:

- Worst-case: $O(n^2)$
- Average-case: $O(n^2)$
- Best-case: $O(n)$ (when the array is already sorted)

4.6.2.0.3 Java Implementation of Bubble Sort:

```
public class BubbleSort {
  public static void bubbleSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
      boolean swapped = false; // Optimization to stop early
      for (int j = 0; j < n - i - 1; j++) {
        if (arr[j] > arr[j + 1]) {
          // Swap elements
          int temp = arr[j];
          arr[j] = arr[j + 1];
          arr[j + 1] = temp;
          swapped = true;
        }
      }
      if (!swapped) break; // Stop if no swaps were made
    }
  }

  public static void main(String[] args) {
    int[] arr = {5, 1, 4, 2, 8};
    bubbleSort(arr);
    for (int num : arr) {
      System.out.print(num + " ");
    }
  }
}
```

Output:

```
1 2 4 5 8
```

4.6.3 Merge Sort: Implementation and Analysis

4.6.3.0.1 What is Merge Sort? Merge Sort is a divide-and-conquer algorithm that splits the array into halves, sorts them recursively, and merges the sorted halves.

4.6.3.0.2 Time Complexity:

- Worst-case: $O(n \log n)$

- Average-case: $O(n \log n)$
- Best-case: $O(n \log n)$

4.6.3.0.3 Java Implementation of Merge Sort:

```
public class MergeSort {
  public static void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
      int mid = (left + right) / 2;

      // Recursively split and sort both halves
      mergeSort(arr, left, mid);
      mergeSort(arr, mid + 1, right);

      // Merge the sorted halves
      merge(arr, left, mid, right);
    }
  }

  private static void merge(int[] arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int[] leftArr = new int[n1];
    int[] rightArr = new int[n2];

    // Copy data to temporary arrays
    System.arraycopy(arr, left, leftArr, 0, n1);
    System.arraycopy(arr, mid + 1, rightArr, 0, n2);

    int i = 0, j = 0, k = left;

    // Merge arrays
    while (i < n1 && j < n2) {
      if (leftArr[i] <= rightArr[j]) {
        arr[k++] = leftArr[i++];
      } else {
        arr[k++] = rightArr[j++];
      }
    }

    // Copy remaining elements
    while (i < n1) arr[k++] = leftArr[i++];
    while (j < n2) arr[k++] = rightArr[j++];
  }

  public static void main(String[] args) {
    int[] arr = {5, 2, 8, 3, 1};
    mergeSort(arr, 0, arr.length - 1);
    for (int num : arr) {
      System.out.print(num + " ");
    }
  }
}
```

Output:


```
1 2 3 5 8
```

4.6.4 Quick Sort: Implementation and Analysis

4.6.4.0.1 What is Quick Sort? Quick Sort is a divide-and-conquer algorithm that selects a pivot element, partitions the array around the pivot, and sorts the subarrays recursively.

4.6.4.0.2 Time Complexity:

- Worst-case: $O(n^2)$ (when the array is already sorted)
- Average-case: $O(n \log n)$
- Best-case: $O(n \log n)$

4.6.4.0.3 Java Implementation of Quick Sort:

```java
public class QuickSort {
  public static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
      int pi = partition(arr, low, high);

      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
    }
  }

  private static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
      if (arr[j] <= pivot) {
        i++;
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
      }
    }

    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;
  }

  public static void main(String[] args) {
    int[] arr = {10, 7, 8, 9, 1, 5};
    quickSort(arr, 0, arr.length - 1);
```



```
    for (int num : arr) {
      System.out.print(num + " ");
    }
  }
}
```

Output:

```
1 5 7 8 9 10
```

4.6.5 Radix Sort: Implementation and Analysis

4.6.5.0.1 What is Radix Sort? Radix Sort is a non-comparative sorting algorithm that processes digits of numbers from the least significant to the most significant place.

4.6.5.0.2 Java Implementation of Radix Sort:

```
import java.util.Arrays;

public class RadixSort {
  public static void radixSort(int[] arr) {
    int max = Arrays.stream(arr).max().getAsInt();
    for (int exp = 1; max / exp > 0; exp *= 10) {
      countingSort(arr, exp);
    }
  }

  private static void countingSort(int[] arr, int exp) {
    int[] output = new int[arr.length];
    int[] count = new int[10];

    for (int value : arr) count[(value / exp) % 10]++;
    for (int i = 1; i < 10; i++) count[i] += count[i - 1];
    for (int i = arr.length - 1; i >= 0; i--) {
      output[count[(arr[i] / exp) % 10] - 1] = arr[i];
      count[(arr[i] / exp) % 10]--;
    }
    System.arraycopy(output, 0, arr, 0, arr.length);
  }

  public static void main(String[] args) {
    int[] arr = {170, 45, 75, 90, 802, 24};
    radixSort(arr);
    for (int num : arr) System.out.print(num + " ");
  }
}
```

Output:

```
24 45 75 90 170 802
```


4.6.6 Summary of Sorting Algorithms

- Bubble Sort: Simple but inefficient for large datasets.
- Merge Sort: Divide-and-conquer with consistent $O(n \log n)$ performance.
- Quick Sort: Efficient for most cases but may degrade to $O(n^2)$.
- Radix Sort: Suitable for integers and works in linear time.

By understanding and implementing these algorithms, developers can optimize sorting operations for different use cases.

4.7 Collections Advanced: Priority Queues and Comparators

Java provides powerful utilities in its Collections Framework for advanced data structure management. This chapter explores:

- `PriorityQueue`: A queue based on heap structures for efficient priority-based ordering.
- Custom Comparators: Defining sorting logic for objects stored in collections.

4.7.1 PriorityQueue for Heap Structures

4.7.1.0.1 What is a Priority Queue? A `PriorityQueue` is a data structure that processes elements based on their priorities rather than their insertion order. It is implemented as a heap:

- By default, it behaves as a Min-Heap, where the smallest element has the highest priority.
- A Comparator can be used to define custom sorting logic.

4.7.1.0.2 Key Features of `PriorityQueue`:

- Elements are ordered based on natural ordering (for objects implementing `Comparable`) or a custom comparator.
- Insertions and deletions are performed in $O(\log n)$ time complexity.
- Not thread-safe (use `PriorityBlockingQueue` for thread safety).

4.7.1.0.3 Example: Basic Usage of `PriorityQueue`

```
import java.util.PriorityQueue;

public class PriorityQueueExample {
  public static void main(String[] args) {
    // Create a PriorityQueue of integers
    PriorityQueue<Integer> pq = new PriorityQueue<>();

    // Add elements to the queue
    pq.add(10);
    pq.add(5);
    pq.add(20);
```



```
    pq.add(1);

    // Retrieve and remove elements (smallest first)
    System.out.println("PriorityQueue elements (ordered by priority):");
    while (!pq.isEmpty()) {
      System.out.println(pq.poll()); // Removes and returns the head of the queue
    }
  }
}
```

Output:

```
PriorityQueue elements (ordered by priority):
1
5
10
20
```

4.7.1.0.4 Explanation:

- Elements are processed in ascending order due to the default Min-Heap implementation.
- `add()` inserts an element into the heap.
- `poll()` retrieves and removes the element with the highest priority (smallest value in this case).

4.7.2 Custom Comparators for Sorting

4.7.2.0.1 What is a Comparator? A Comparator defines custom sorting logic for objects. It is used when:

- Objects do not implement the `Comparable` interface.
- Sorting needs to follow a custom order instead of natural ordering.

4.7.2.0.2 Defining a Custom Comparator for `PriorityQueue`

```
import java.util.PriorityQueue;
import java.util.Comparator;

// Custom Comparator for descending order
class DescendingOrderComparator implements Comparator<Integer> {
  @Override
  public int compare(Integer a, Integer b) {
    return b - a; // Sort in descending order
  }
```



```
}

public class CustomComparatorExample {
  public static void main(String[] args) {
    // PriorityQueue with custom comparator
    PriorityQueue<Integer> pq = new PriorityQueue<>(new DescendingOrderComparator());

    // Add elements to the queue
    pq.add(10);
    pq.add(5);
    pq.add(20);
    pq.add(1);

    // Retrieve and remove elements (largest first)
    System.out.println("PriorityQueue elements (descending order):");
    while (!pq.isEmpty()) {
      System.out.println(pq.poll());
    }
  }
}
```

Output:

```
PriorityQueue elements (descending order):
20
10
5
1
```

4.7.2.0.3 Explanation:

- The custom comparator sorts elements in descending order.
- The comparator logic is defined in the `compare()` method (`b - a`).

4.7.2.0.4 Custom Comparators for Complex Objects Sorting objects in a priority queue requires custom logic for comparisons.

```
import java.util.PriorityQueue;
import java.util.Comparator;

// Define a Student class
class Student {
  String name;
  int score;

  public Student(String name, int score) {
    this.name = name;
    this.score = score;
  }
```



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

// Custom Comparator to sort students by score
class StudentScoreComparator implements Comparator<Student> {
  @Override
  public int compare(Student s1, Student s2) {
    return s2.score - s1.score; // Sort by descending score
  }
}

public class CustomObjectComparator {
  public static void main(String[] args) {
    // PriorityQueue for Student objects with custom comparator
    PriorityQueue<Student> pq = new PriorityQueue<>(new StudentScoreComparator());

    // Add students to the queue
    pq.add(new Student("Alice", 85));
    pq.add(new Student("Bob", 95));
    pq.add(new Student("Charlie", 75));

    // Process students based on score
    System.out.println("Students ordered by score (highest first):");
    while (!pq.isEmpty()) {
      System.out.println(pq.poll());
    }
  }
}
```

Output:

```
Students ordered by score (highest first):
Bob: 95
Alice: 85
Charlie: 75
```

4.7.2.0.5 Explanation:

- The custom comparator sorts `Student` objects by descending scores.
- `compare()` defines the sorting logic.
- `toString()` is overridden for readable output.

4.7.3 Best Practices for Using Priority Queues and Comparators

4.7.3.0.1 Best Practices for `PriorityQueue`

- Use the default natural ordering for basic types like integers and strings.
- Define custom comparators for complex objects.
- Always check for null values as `PriorityQueue` does not allow null elements.
- Use Min-Heap (default behavior) or Max-Heap by providing appropriate comparators.

4.7.3.0.2 Best Practices for Comparators

- Implement the `compare()` method carefully to avoid logical errors.
- Use lambda expressions or method references (Java 8+) for simple comparators.
- Ensure comparators are consistent with `equals()` to avoid unexpected behavior.

4.7.3.0.3 Example: Simplifying Comparators with Lambda Expressions

```
import java.util.PriorityQueue;

public class LambdaComparatorExample {
  public static void main(String[] args) {
    // PriorityQueue with lambda expression for descending order
    PriorityQueue<Integer> pq = new PriorityQueue<>((a, b) -> b - a);

    pq.add(10);
    pq.add(5);
    pq.add(20);
    pq.add(1);

    System.out.println("Elements in descending order:");
    while (!pq.isEmpty()) {
      System.out.println(pq.poll());
    }
  }
}
```

Output:

```
Elements in descending order:
20
10
5
1
```


4.7.3.0.4 Summary of Priority Queues and Comparators

- `PriorityQueue` is a heap-based data structure that processes elements based on priority.
- The default implementation is a Min-Heap that orders elements in natural ascending order.
- Custom comparators allow flexible and dynamic sorting for primitive and complex types.
- Use lambda expressions for concise and readable comparator logic.
- Always ensure comparators are consistent and properly tested for correctness.

By mastering `PriorityQueue` and custom comparators, developers can efficiently manage and sort data in advanced Java applications.

4.8 Native Java Sorting and Searching Algorithms in Java

Sorting and searching are fundamental operations in computer science, and Java provides both built-in and manual implementations for these algorithms. This chapter explains:

- Sorting algorithms: Bubble Sort, Merge Sort, and Quick Sort.
- Searching algorithms: Binary Search and Linear Search.

We explore their implementation, time complexities, and use cases.

4.8.1 Implementing Sorting Algorithms

4.8.1.0.1 Bubble Sort Bubble Sort is a simple comparison-based algorithm where adjacent elements are repeatedly compared and swapped if needed.

4.8.1.0.2 Java Implementation of Bubble Sort

```
public class BubbleSort {
  public static void bubbleSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - i - 1; j++) {
        if (arr[j] > arr[j + 1]) {
          // Swap adjacent elements
          int temp = arr[j];
          arr[j] = arr[j + 1];
          arr[j + 1] = temp;
        }
      }
    }
  }

  public static void main(String[] args) {
    int[] arr = {5, 3, 8, 4, 2};
    bubbleSort(arr);
    System.out.print("Sorted Array: ");
    for (int num : arr) {
      System.out.print(num + " ");
    }
  }
}
```

Output:

```
Sorted Array: 2 3 4 5 8
```


4.8.1.0.3 Time Complexity:

- Worst-case: $O(n^2)$
- Best-case: $O(n)$ (when the array is already sorted)

4.8.1.0.4 Merge Sort Merge Sort is a divide-and-conquer algorithm that splits the array into halves, sorts each half, and merges the results.

4.8.1.0.5 Java Implementation of Merge Sort

```
public class MergeSort {
  public static void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
      int mid = left + (right - left) / 2;

      // Recursively divide and sort
      mergeSort(arr, left, mid);
      mergeSort(arr, mid + 1, right);

      // Merge the sorted halves
      merge(arr, left, mid, right);
    }
  }

  private static void merge(int[] arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int[] leftArr = new int[n1];
    int[] rightArr = new int[n2];

    for (int i = 0; i < n1; i++) leftArr[i] = arr[left + i];
    for (int j = 0; j < n2; j++) rightArr[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
      if (leftArr[i] <= rightArr[j]) {
        arr[k++] = leftArr[i++];
      } else {
        arr[k++] = rightArr[j++];
      }
    }

    while (i < n1) arr[k++] = leftArr[i++];
    while (j < n2) arr[k++] = rightArr[j++];
  }

  public static void main(String[] args) {
    int[] arr = {12, 11, 13, 5, 6, 7};
    mergeSort(arr, 0, arr.length - 1);
    System.out.print("Sorted Array: ");
    for (int num : arr) System.out.print(num + " ");
```



```
  }
}
```

Output:

```
Sorted Array: 5 6 7 11 12 13
```

4.8.1.0.6 Time Complexity:

- Worst-case, Best-case, and Average-case: $O(n \log n)$

4.8.1.0.7 Quick Sort Quick Sort is another divide-and-conquer algorithm that selects a pivot element, partitions the array, and sorts the partitions recursively.

4.8.1.0.8 Java Implementation of Quick Sort

```
public class QuickSort {
  public static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
      int pi = partition(arr, low, high);
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
    }
  }

  private static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
      if (arr[j] <= pivot) {
        i++;
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
      }
    }

    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;
  }

  public static void main(String[] args) {
    int[] arr = {10, 7, 8, 9, 1, 5};
    quickSort(arr, 0, arr.length - 1);
    System.out.print("Sorted Array: ");
    for (int num : arr) System.out.print(num + " ");
  }
}
```


Output:

```
Sorted Array: 1 5 7 8 9 10
```

4.8.1.0.9 Time Complexity:

- Worst-case: $O(n^2)$ (when the array is already sorted).
- Best-case and Average-case: $O(n \log n)$.

4.8.2 Searching Algorithms

4.8.2.0.1 Linear Search Linear Search checks each element sequentially. It works for unsorted arrays.

```
public class LinearSearch {
  public static int linearSearch(int[] arr, int key) {
    for (int i = 0; i < arr.length; i++) {
      if (arr[i] == key) return i;
    }
    return -1;
  }

  public static void main(String[] args) {
    int[] arr = {10, 20, 30, 40, 50};
    int key = 30;

    int result = linearSearch(arr, key);
    System.out.println(key + " found at index: " + result);
  }
}
```

Output:

```
30 found at index: 2
```

4.8.2.0.2 Binary Search Binary Search works only on sorted arrays. It divides the search range in half repeatedly.

```
public class BinarySearch {
  public static int binarySearch(int[] arr, int key) {
    int low = 0, high = arr.length - 1;

    while (low <= high) {
      int mid = low + (high - low) / 2;

      if (arr[mid] == key) return mid;
      else if (arr[mid] < key) low = mid + 1;
      else high = mid - 1;
```



```
    }
    return -1;
  }

  public static void main(String[] args) {
    int[] arr = {1, 2, 3, 4, 5, 6};
    int key = 4;

    int result = binarySearch(arr, key);
    System.out.println(key + " found at index: " + result);
  }
}
```

Output:

```
4 found at index: 3
```

4.8.2.0.3 Time Complexity:

- Worst-case: $O(\log n)$.
- Best-case: $O(1)$ (when the key is at the middle).

4.8.3 Summary of Sorting and Searching Algorithms

- Bubble Sort: Simple but inefficient with $O(n^2)$ time complexity.
- Merge Sort: Divide-and-conquer with $O(n \log n)$ time complexity in all cases.
- Quick Sort: Efficient for most cases with $O(n \log n)$, but can degrade to $O(n^2)$.
- Linear Search: Sequential search with $O(n)$ time complexity.
- Binary Search: Efficient search for sorted arrays with $O(\log n)$ complexity.

4.9 Data Structures: Stacks, Queues, and Linked Lists

Data structures like Stacks, Queues, and Linked Lists are fundamental for managing and organizing data. This chapter explains:

- Implementing and using Stacks and Queues.
- Understanding and implementing Singly Linked Lists and Doubly Linked Lists.

4.9.1 Implementing and Using Stacks

4.9.1.0.1 What is a Stack? A Stack is a linear data structure that follows the LIFO (Last-In, First-Out) principle. The most recently added element is removed first.

4.9.1.0.2 Key Stack Operations

- Push: Add an element to the top of the stack.
- Pop: Remove and return the top element.
- Peek: Return the top element without removing it.

4.9.1.0.3 Example: Implementing a Stack Using an Array

```
class Stack {
  private int[] stack;
  private int top;
  private int capacity;

  public Stack(int size) {
    capacity = size;
    stack = new int[capacity];
    top = -1;
  }

  // Push an element onto the stack
  public void push(int value) {
    if (top == capacity - 1) {
      System.out.println("Stack Overflow");
      return;
    }
    stack[++top] = value;
  }

  // Pop the top element from the stack
```



```
  public int pop() {
    if (isEmpty()) {
      System.out.println("Stack Underflow");
      return -1;
    }
    return stack[top--];
  }

  // Peek at the top element
  public int peek() {
    if (isEmpty()) {
      System.out.println("Stack is empty");
      return -1;
    }
    return stack[top];
  }

  // Check if the stack is empty
  public boolean isEmpty() {
    return top == -1;
  }
}

public class StackExample {
  public static void main(String[] args) {
    Stack stack = new Stack(5);

    stack.push(10);
    stack.push(20);
    stack.push(30);

    System.out.println("Top element: " + stack.peek());
    System.out.println("Popped element: " + stack.pop());
    System.out.println("Top element after pop: " + stack.peek());
  }
}
```

Output:

```
Top element: 30
Popped element: 30
Top element after pop: 20
```

4.9.2 Implementing and Using Queues

4.9.2.0.1 What is a Queue? A Queue is a linear data structure that follows the FIFO (First-In, First-Out) principle. The first element added is the first to be removed.

4.9.2.0.2 Key Queue Operations

- Enqueue: Add an element to the rear of the queue.

- Dequeue: Remove and return the front element.
- Peek: Return the front element without removing it.

4.9.2.0.3 Example: Implementing a Queue Using an Array

```
class Queue {
  private int[] queue;
  private int front, rear, size, capacity;

  public Queue(int capacity) {
    this.capacity = capacity;
    queue = new int[capacity];
    front = 0;
    rear = -1;
    size = 0;
  }

  // Enqueue operation
  public void enqueue(int value) {
    if (size == capacity) {
      System.out.println("Queue Overflow");
      return;
    }
    rear = (rear + 1) % capacity;
    queue[rear] = value;
    size++;
  }

  // Dequeue operation
  public int dequeue() {
    if (isEmpty()) {
      System.out.println("Queue Underflow");
      return -1;
    }
    int value = queue[front];
    front = (front + 1) % capacity;
    size--;
    return value;
  }

  // Peek at the front element
  public int peek() {
    if (isEmpty()) {
      System.out.println("Queue is empty");
      return -1;
    }
    return queue[front];
  }

  // Check if the queue is empty
  public boolean isEmpty() {
    return size == 0;
  }
}
```



```
public class QueueExample {
  public static void main(String[] args) {
    Queue queue = new Queue(5);

    queue.enqueue(10);
    queue.enqueue(20);
    queue.enqueue(30);

    System.out.println("Front element: " + queue.peek());
    System.out.println("Dequeued element: " + queue.dequeue());
    System.out.println("Front element after dequeue: " + queue.peek());
  }
}
```

Output:

```
Front element: 10
Dequeued element: 10
Front element after dequeue: 20
```

4.9.3 Singly and Doubly Linked Lists

4.9.3.0.1 What is a Linked List? A Linked List is a linear data structure where elements (nodes) are linked together using pointers. Each node contains:

- Data: The value stored in the node.
- Pointer: A reference to the next node.

4.9.3.0.2 Singly Linked List Implementation

```
class Node {
  int data;
  Node next;

  public Node(int data) {
    this.data = data;
    this.next = null;
  }
}

class SinglyLinkedList {
  private Node head;

  // Insert a new node at the end
  public void insert(int data) {
    Node newNode = new Node(data);
    if (head == null) {
      head = newNode;
      return;
    }
```



```
    Node current = head;
    while (current.next != null) {
      current = current.next;
    }
    current.next = newNode;
  }

  // Display the linked list
  public void display() {
    Node current = head;
    while (current != null) {
      System.out.print(current.data + " -> ");
      current = current.next;
    }
    System.out.println("null");
  }
}

public class SinglyLinkedListExample {
  public static void main(String[] args) {
    SinglyLinkedList list = new SinglyLinkedList();

    list.insert(10);
    list.insert(20);
    list.insert(30);

    System.out.print("Linked List: ");
    list.display();
  }
}
```

Output:

```
Linked List: 10 -> 20 -> 30 -> null
```

4.9.3.0.3 Doubly Linked List Implementation

```
class DoublyNode {
  int data;
  DoublyNode prev, next;

  public DoublyNode(int data) {
    this.data = data;
    this.prev = this.next = null;
  }
}

class DoublyLinkedList {
  private DoublyNode head;

  // Insert a new node at the end
  public void insert(int data) {
    DoublyNode newNode = new DoublyNode(data);
    if (head == null) {
      head = newNode;
      return;
```



```
    }
    DoublyNode current = head;
    while (current.next != null) {
      current = current.next;
    }
    current.next = newNode;
    newNode.prev = current;
  }

  // Display the linked list forward
  public void displayForward() {
    DoublyNode current = head;
    while (current != null) {
      System.out.print(current.data + " -> ");
      current = current.next;
    }
    System.out.println("null");
  }
}

public class DoublyLinkedListExample {
  public static void main(String[] args) {
    DoublyLinkedList list = new DoublyLinkedList();

    list.insert(10);
    list.insert(20);
    list.insert(30);

    System.out.print("Doubly Linked List (Forward): ");
    list.displayForward();
  }
}
```

Output:

```
Doubly Linked List (Forward): 10 -> 20 -> 30 -> null
```

4.9.4 Comparison of Stacks, Queues, and Linked Lists

- Stack: Follows LIFO (Last-In, First-Out) and supports `push`, `pop`, and `peek`.
- Queue: Follows FIFO (First-In, First-Out) and supports `enqueue`, `dequeue`, and `peek`.
- Singly Linked List: Each node points to the next node, providing dynamic memory allocation.
- Doubly Linked List: Each node has references to both the previous and next nodes, enabling bidirectional traversal.

4.10 Implementing Binary Trees and Graphs

Binary Trees and Graphs are essential data structures used in various applications, including search algorithms, networking, and hierarchical data representation. This chapter focuses on:

- Building and traversing binary trees.
- Representing and traversing graphs.

4.10.1 Building and Traversing Binary Trees

4.10.1.0.1 What is a Binary Tree? A Binary Tree is a hierarchical data structure where each node has at most two children:

- Left Child
- Right Child

4.10.1.0.2 Binary Tree Node Structure Each node in a binary tree consists of:

- Data: The value stored in the node.
- Left: Pointer to the left child.
- Right: Pointer to the right child.

4.10.1.0.3 Example: Building a Binary Tree and Preorder Traversal

```
class TreeNode {
  int data;
  TreeNode left, right;

  public TreeNode(int data) {
    this.data = data;
    this.left = this.right = null;
  }
}

class BinaryTree {
  TreeNode root;

  // Preorder Traversal: Root -> Left -> Right
  public void preorder(TreeNode node) {
    if (node == null) return;
```



```
    System.out.print(node.data + " ");
    preorder(node.left);
    preorder(node.right);
  }
}

public class BinaryTreeExample {
  public static void main(String[] args) {
    BinaryTree tree = new BinaryTree();

    // Building the binary tree
    tree.root = new TreeNode(1);
    tree.root.left = new TreeNode(2);
    tree.root.right = new TreeNode(3);
    tree.root.left.left = new TreeNode(4);
    tree.root.left.right = new TreeNode(5);

    System.out.print("Preorder Traversal: ");
    tree.preorder(tree.root);
  }
}
```

Output:

```
Preorder Traversal: 1 2 4 5 3
```

4.10.1.0.4 Binary Tree Traversals

Binary trees can be traversed in three main ways:

- Preorder (Root -¿ Left -¿ Right)
- Inorder (Left -¿ Root -¿ Right)
- Postorder (Left -¿ Right -¿ Root)

4.10.1.0.5 Example: Inorder and Postorder Traversals

```
class BinaryTreeTraversal {
  TreeNode root;

  // Inorder Traversal: Left -> Root -> Right
  public void inorder(TreeNode node) {
    if (node == null) return;

    inorder(node.left);
    System.out.print(node.data + " ");
    inorder(node.right);
  }

  // Postorder Traversal: Left -> Right -> Root
  public void postorder(TreeNode node) {
    if (node == null) return;
```



```
    postorder(node.left);
    postorder(node.right);
    System.out.print(node.data + " ");
  }
}

public class TraversalExample {
  public static void main(String[] args) {
    BinaryTreeTraversal tree = new BinaryTreeTraversal();

    tree.root = new TreeNode(1);
    tree.root.left = new TreeNode(2);
    tree.root.right = new TreeNode(3);
    tree.root.left.left = new TreeNode(4);
    tree.root.left.right = new TreeNode(5);

    System.out.print("Inorder Traversal: ");
    tree.inorder(tree.root);
    System.out.println();

    System.out.print("Postorder Traversal: ");
    tree.postorder(tree.root);
  }
}
```

Output:

```
Inorder Traversal: 4 2 5 1 3
Postorder Traversal: 4 5 2 3 1
```

4.10.2 Graph Representation and Traversal

4.10.2.0.1 What is a Graph? A Graph is a collection of vertices (or nodes) and edges (connections between vertices). Graphs can be:

- Directed: Edges have a direction (e.g., A to B).
- Undirected: Edges are bidirectional (e.g., A to B or B to A).
- Weighted: Edges have weights or costs.

4.10.2.0.2 Graph Representation in Java Graphs are commonly represented using:

- Adjacency Matrix: A 2D array where `matrix[i][j]` represents the edge between nodes i and j.
- Adjacency List: An array or list of lists where each index corresponds to a vertex and contains a list of its adjacent vertices.

4.10.2.0.3 Example: Graph Representation Using Adjacency List

```java
import java.util.*;

class Graph {
  private Map<Integer, List<Integer>> adjList;

  public Graph() {
    adjList = new HashMap<>();
  }

  // Add an edge to the graph
  public void addEdge(int source, int destination) {
    adjList.putIfAbsent(source, new ArrayList<>());
    adjList.get(source).add(destination);

    adjList.putIfAbsent(destination, new ArrayList<>()); // For undirected graph
    adjList.get(destination).add(source);
  }

  // Display the graph
  public void display() {
    for (int vertex : adjList.keySet()) {
      System.out.print(vertex + " -> " + adjList.get(vertex));
      System.out.println();
    }
  }
}

public class GraphExample {
  public static void main(String[] args) {
    Graph graph = new Graph();

    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);
    graph.addEdge(3, 4);

    System.out.println("Graph (Adjacency List):");
    graph.display();
  }
}
```

Output:

```
Graph (Adjacency List):
1 -> [2, 3]
2 -> [1, 4]
3 -> [1, 4]
4 -> [2, 3]
```

4.10.2.0.4 Depth-First Search (DFS) for Graphs

DFS explores as far down a branch as possible before backtracking.


```
class DFS {
  private Map<Integer, List<Integer>> adjList = new HashMap<>();

  public void addEdge(int source, int destination) {
    adjList.putIfAbsent(source, new ArrayList<>());
    adjList.get(source).add(destination);
  }

  public void dfs(int start) {
    Set<Integer> visited = new HashSet<>();
    dfsHelper(start, visited);
  }

  private void dfsHelper(int vertex, Set<Integer> visited) {
    if (visited.contains(vertex)) return;

    System.out.print(vertex + " ");
    visited.add(vertex);

    for (int neighbor : adjList.getOrDefault(vertex, new ArrayList<>())) {
      dfsHelper(neighbor, visited);
    }
  }
}

public class DFSExample {
  public static void main(String[] args) {
    DFS graph = new DFS();

    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);
    graph.addEdge(3, 4);

    System.out.print("DFS Traversal: ");
    graph.dfs(1);
  }
}
```

Output:

```
DFS Traversal: 1 2 4 3
```

4.10.2.0.5 Breadth-First Search (BFS) for Graphs

BFS explores all neighbors at the current depth before moving deeper.

```
class BFS {
  private Map<Integer, List<Integer>> adjList = new HashMap<>();

  public void addEdge(int source, int destination) {
    adjList.putIfAbsent(source, new ArrayList<>());
    adjList.get(source).add(destination);
  }

  public void bfs(int start) {
```



```
    Set<Integer> visited = new HashSet<>();
    Queue<Integer> queue = new LinkedList<>();
    queue.add(start);
    visited.add(start);

    while (!queue.isEmpty()) {
      int vertex = queue.poll();
      System.out.print(vertex + " ");

      for (int neighbor : adjList.getOrDefault(vertex, new ArrayList<>())) {
        if (!visited.contains(neighbor)) {
          visited.add(neighbor);
          queue.add(neighbor);
        }
      }
    }
  }
}

public class BFSExample {
  public static void main(String[] args) {
    BFS graph = new BFS();

    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);
    graph.addEdge(3, 4);

    System.out.print("BFS Traversal: ");
    graph.bfs(1);
  }
}
```

Output:

```
BFS Traversal: 1 2 3 4
```

4.10.3 Summary of Binary Trees and Graphs

- Binary Trees are hierarchical structures where each node has at most two children. Traversals include preorder, inorder, and postorder.
- Graphs can be represented using adjacency lists or matrices and traversed using DFS or BFS.
- DFS explores nodes deeply, while BFS explores nodes level by level.

Chapter 5

Java Standard Library and Utils

5.1 Java Standard Library: Math, Date, and Utility Classes

The Java Standard Library provides powerful tools for mathematical computations, date and time manipulation, and various utility tasks. This chapter focuses on:

- The `Math` class for mathematical operations.
- Working with dates using `java.util.Date` and the modern `java.time` API.
- Utility classes like `Random` for generating random values and `Scanner` for user input.

5.1.1 Mathematical Operations with the `Math` Class

5.1.1.0.1 The `Math` Class Overview The `Math` class in Java provides static methods for mathematical operations like trigonometry, logarithms, rounding, and random number generation.

5.1.1.0.2 Example: Common Math Operations

```
public class MathExample {
  public static void main(String[] args) {
    double a = 25;

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

    // Power
    System.out.println("2 raised to 3: " + Math.pow(2, 3));

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

    // Trigonometric operations
    System.out.println("Sine of 90 degrees: " + Math.sin(Math.toRadians(90)));
    System.out.println("Cosine of 0 degrees: " + Math.cos(Math.toRadians(0)));

    // Rounding
    System.out.println("Ceiling of 4.2: " + Math.ceil(4.2));
    System.out.println("Floor of 4.7: " + Math.floor(4.7));
    System.out.println("Round 4.5: " + Math.round(4.5));

    // Random number between 0 and 1
    System.out.println("Random number: " + Math.random());
  }
}
```

Output:


```
Square root of 25: 5.0
2 raised to 3: 8.0
Absolute value of -10: 10
Sine of 90 degrees: 1.0
Cosine of 0 degrees: 1.0
Ceiling of 4.2: 5.0
Floor of 4.7: 4.0
Round 4.5: 5
Random number: 0.675342
```

5.1.2 Working with Dates: `java.util.Date` and `java.time` API

5.1.2.0.1 Legacy `java.util.Date` The `Date` class is part of the older date and time API and is now largely replaced by the modern `java.time` API.

5.1.2.0.2 Example: Using `java.util.Date`

```
import java.util.Date;

public class LegacyDateExample {
  public static void main(String[] args) {
    // Current date and time
    Date currentDate = new Date();
    System.out.println("Current Date: " + currentDate);

    // Time in milliseconds since January 1, 1970
    long timeInMillis = currentDate.getTime();
    System.out.println("Milliseconds since epoch: " + timeInMillis);
  }
}
```

Output:

```
Current Date: Mon Jan 01 12:00:00 GMT 2024
Milliseconds since epoch: 1704120000000
```

5.1.2.0.3 Modern `java.time` API (Java 8+) The `java.time` API provides a more robust and readable way to handle date and time.

5.1.2.0.4 Example: Using `LocalDate`, `LocalTime`, and `LocalDateTime`

```
import java.time.*;
```



```
public class ModernDateExample {
  public static void main(String[] args) {
    // Current Date
    LocalDate today = LocalDate.now();
    System.out.println("Current Date: " + today);

    // Current Time
    LocalTime now = LocalTime.now();
    System.out.println("Current Time: " + now);

    // Current Date and Time
    LocalDateTime dateTime = LocalDateTime.now();
    System.out.println("Current Date and Time: " + dateTime);

    // Specific Date and Time
    LocalDate specificDate = LocalDate.of(2024, 1, 1);
    LocalTime specificTime = LocalTime.of(10, 30);
    System.out.println("Specific Date: " + specificDate);
    System.out.println("Specific Time: " + specificTime);

    // Formatting
    System.out.println("Formatted Date: " + dateTime.toLocalDate());
  }
}
```

Output:

```
Current Date: 2024-01-01
Current Time: 12:34:56.123
Current Date and Time: 2024-01-01T12:34:56.123
Specific Date: 2024-01-01
Specific Time: 10:30
Formatted Date: 2024-01-01
```

5.1.3 Random, Scanner, and Other Utility Classes

5.1.3.0.1 Generating Random Numbers Using `Random` The `Random` class in the `java.util` package generates random values.

```
import java.util.Random;

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

    // Random integers
    System.out.println("Random Integer: " + random.nextInt(100)); // Between 0 and 99

    // Random doubles
    System.out.println("Random Double: " + random.nextDouble());

    // Random booleans
```



```
    System.out.println("Random Boolean: " + random.nextBoolean());
  }
}
```

Output:

```
Random Integer: 45
Random Double: 0.783423
Random Boolean: true
```

5.1.3.0.2 Using `Scanner` for User Input

The `Scanner` class allows reading input from the user.

```
import java.util.Scanner;

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

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

    System.out.print("Enter your age: ");
    int age = scanner.nextInt();

    System.out.println("Hello, " + name + "! You are " + age + " years old.");
    scanner.close();
  }
}
```

Output:

```
Enter your name: Alice
Enter your age: 25
Hello, Alice! You are 25 years old.
```

5.1.4 Best Practices for Utility Classes

- Use the `java.time` API instead of the older `java.util.Date` for date/time operations.
- Use the `Random` class or `ThreadLocalRandom` for generating random numbers.
- Always close resources like `Scanner` to prevent resource leaks.
- Use `Math` for mathematical operations instead of manual implementations.

5.2 File Handling and Input/Output (I/O)

File handling in Java is the process of reading from and writing to files stored on a file system. Java provides several APIs for file handling, including the classic `File` API, streams for reading and writing data, and advanced I/O using NIO (New I/O) and NIO.2.

5.2.1 Working with Files Using the File API

5.2.1.0.1 Introduction to the `File` API The `File` class (in the `java.io` package) provides methods to work with file and directory operations such as creating, deleting, and checking file information.

5.2.1.0.2 Example: Creating and Deleting Files

```
import java.io.File;
import java.io.IOException;

public class FileAPIExample {
  public static void main(String[] args) {
    try {
      File file = new File("example.txt");

      // Create a new file
      if (file.createNewFile()) {
        System.out.println("File created: " + file.getName());
      } else {
        System.out.println("File already exists.");
      }

      // Check file properties
      System.out.println("Absolute Path: " + file.getAbsolutePath());
      System.out.println("Is File Writable? " + file.canWrite());
      System.out.println("Is File Readable? " + file.canRead());

      // Delete the file
      if (file.delete()) {
        System.out.println("File deleted successfully.");
      } else {
        System.out.println("Failed to delete the file.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

Output:


```
File created: example.txt
Absolute Path: /path/to/example.txt
Is File Writable? true
Is File Readable? true
File deleted successfully.
```

Explanation:

- `createNewFile()` creates a new file.
- `delete()` deletes the file.
- File properties such as read/write permissions are checked using `canRead()` and `canWrite()`.

5.2.2 Reading and Writing Text and Binary Data

5.2.2.0.1 Reading and Writing Text Files Java provides `FileReader` and `FileWriter` for character-based file I/O.

5.2.2.0.2 Writing to a File Using `FileWriter`:

```
import java.io.FileWriter;
import java.io.IOException;

public class WriteTextFile {
  public static void main(String[] args) {
    try (FileWriter writer = new FileWriter("output.txt")) {
      writer.write("Hello, World!\n");
      writer.write("Welcome to Java File I/O.");
      System.out.println("Data written to file.");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

5.2.2.0.3 Reading from a File Using `FileReader`:

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

public class ReadTextFile {
  public static void main(String[] args) {
    try (FileReader reader = new FileReader("output.txt")) {
      int character;
      while ((character = reader.read()) != -1) {
        System.out.print((char) character);
```



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

Output:

```
Hello, World!
Welcome to Java File I/O.
```

5.2.2.0.4 Reading and Writing Binary Files Use `FileInputStream` and `FileOutputStream` for binary data (e.g., images or executable files).

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class BinaryFileExample {
  public static void main(String[] args) {
    try (FileInputStream input = new FileInputStream("input.jpg");
    FileOutputStream output = new FileOutputStream("copy.jpg")) {

      int byteData;
      while ((byteData = input.read()) != -1) {
        output.write(byteData);
      }
      System.out.println("File copied successfully.");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Explanation:

- `FileInputStream` reads binary data byte by byte.
- `FileOutputStream` writes binary data to a new file.

5.2.3 NIO and NIO.2 for Advanced I/O

5.2.3.0.1 What is NIO (New I/O)? The NIO (introduced in Java 1.4) and NIO.2 (introduced in Java 7) APIs provide efficient, non-blocking I/O operations. The key classes include:

- `Path`: Represents file or directory paths.

- `Files`: Contains utility methods for file manipulation.
- `Channels`: Used for non-blocking I/O.
- `Buffers`: Used for data storage during I/O operations.

5.2.3.0.2 Example: Using `Path` and `Files` for File Operations

```
import java.nio.file.*;
import java.io.IOException;

public class NIOFileExample {
  public static void main(String[] args) {
    Path filePath = Paths.get("nio_example.txt");

    try {
      // Write to a file using NIO
      Files.write(filePath, "Hello, NIO!\nWelcome to NIO.2.".getBytes());
      System.out.println("File written successfully.");

      // Read from a file using NIO
      String content = Files.readString(filePath);
      System.out.println("File Content:");
      System.out.println(content);

      // Delete the file
      Files.deleteIfExists(filePath);
      System.out.println("File deleted successfully.");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Output:

```
File written successfully.
File Content:
Hello, NIO!
Welcome to NIO.2.
File deleted successfully.
```

5.2.3.0.3 Using Buffers and Channels for Efficient I/O

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.*;

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



```
    Path path = Paths.get("channel_example.txt");

    try (FileChannel channel = FileChannel.open(path, StandardOpenOption.CREATE, StandardOpenOption.WRITE)) {
      String data = "Using FileChannel for NIO file operations.";
      ByteBuffer buffer = ByteBuffer.allocate(1024);
      buffer.put(data.getBytes());

      buffer.flip(); // Prepare buffer for writing
      channel.write(buffer);
      System.out.println("Data written using FileChannel.");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Explanation:

- `FileChannel` is a part of the NIO API and provides efficient, non-blocking file I/O.
- `ByteBuffer` temporarily stores data before writing or reading.

5.2.4 Best Practices for File Handling and I/O

- Always close streams, readers, and writers using the try-with-resources statement.
- Use NIO for improved performance and non-blocking I/O operations.
- Avoid reading large files into memory at once; process data in chunks.
- Use `BufferedReader` and `BufferedWriter` for efficient text I/O.
- Validate file paths and handle exceptions using appropriate error messages.
- The classic `File` API is used for basic file operations like creating, deleting, and checking properties.
- Streams (`FileReader`, `FileWriter`, `FileInputStream`, `FileOutputStream`) handle text and binary data.
- NIO and NIO.2 provide advanced file I/O with features like non-blocking operations, file channels, and utility methods in the `Files` class.
- Use modern APIs like `Path`, `Files`, and `FileChannel` for efficient and scalable file handling.

Chapter 6

Concurrency in Java

6.1 Multithreading and Concurrency

Multithreading and concurrency in Java allow programs to perform multiple tasks simultaneously, improving responsiveness and efficiency. Threads enable parallel execution of code, while Java's concurrency tools manage thread execution safely and efficiently.

This chapter explores the thread lifecycle, thread management, executors, thread pools, and asynchronous programming using `Callable` and `Future`.

6.1.1 Thread Lifecycle and Management

6.1.1.0.1 What is a Thread? A thread is the smallest unit of execution in a program. Multithreading allows multiple threads to run concurrently, enabling tasks to execute in parallel.

6.1.1.0.2 Thread Lifecycle A thread in Java goes through the following states:

- **New**: Thread is created but not started.
- **Runnable**: Thread is ready to run but waiting for CPU allocation.
- **Running**: Thread is executing.
- **Blocked/Waiting**: Thread is waiting for a resource or signal.
- **Terminated**: Thread has completed execution or stopped.

6.1.1.0.3 Creating Threads: Extending `Thread` Class

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

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

    t1.start(); // Start the first thread
    t2.start(); // Start the second thread
  }
}
```

Output:


```
Thread is running: Thread-0
Thread is running: Thread-1
```

6.1.1.0.4 Creating Threads: Implementing `Runnable` Interface

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

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();
  }
}
```

6.1.1.0.5 Choosing Between `Thread` and `Runnable`:

- Use `Thread` when directly extending the thread class.
- Use `Runnable` for better flexibility since Java does not support multiple inheritance.

6.1.2 Executors and Thread Pools

6.1.2.0.1 What is the Executor Framework? The Executor Framework (introduced in Java 5) provides a high-level API to manage threads efficiently. It abstracts thread creation, execution, and termination using thread pools.

6.1.2.0.2 Thread Pools Thread pools manage a group of reusable threads to execute tasks efficiently. Common types include:

- `FixedThreadPool`: A pool with a fixed number of threads.
- `CachedThreadPool`: A pool that creates new threads as needed.
- `SingleThreadExecutor`: A single-threaded executor.
- `ScheduledThreadPool`: A pool that schedules tasks with delays.

6.1.2.0.3 Example: Using `ExecutorService` with Fixed Thread Pool

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Task implements Runnable {
  private final int taskId;

  public Task(int taskId) {
    this.taskId = taskId;
  }

  @Override
  public void run() {
    System.out.println("Executing Task " + taskId + " by " + Thread.currentThread().getName());
  }
}

public class ExecutorExample {
  public static void main(String[] args) {
    ExecutorService executor = Executors.newFixedThreadPool(3);

    for (int i = 1; i <= 5; i++) {
      executor.submit(new Task(i));
    }

    executor.shutdown(); // Gracefully shut down the executor
  }
}
```

Output:

```
Executing Task 1 by pool-1-thread-1
Executing Task 2 by pool-1-thread-2
Executing Task 3 by pool-1-thread-3
Executing Task 4 by pool-1-thread-1
Executing Task 5 by pool-1-thread-2
```

Explanation:

- `Executors.newFixedThreadPool(3)` creates a pool of 3 threads.
- Tasks are submitted to the executor, which executes them using available threads.
- `shutdown()` ensures that no new tasks are accepted, but existing tasks complete execution.

6.1.3 Callable, Future, and Asynchronous Tasks

6.1.3.0.1 What is `Callable`?

`Callable` is a functional interface in Java that allows a thread to return a result or throw a checked exception.

6.1.3.0.2 What is `Future`?

`Future` represents the result of an asynchronous computation. It provides methods to check if a task is complete and retrieve the result.

6.1.3.0.3 Example: Using `Callable` and `Future`

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class FactorialTask implements Callable<Long> {
  private final int number;

  public FactorialTask(int number) {
    this.number = number;
  }

  @Override
  public Long call() throws Exception {
    long result = 1;
    for (int i = 1; i <= number; i++) {
      result *= i;
      Thread.sleep(100); // Simulate computation
    }
    return result;
  }
}

public class CallableExample {
  public static void main(String[] args) throws Exception {
    ExecutorService executor = Executors.newSingleThreadExecutor();

    // Submit Callable task
    Future<Long> future = executor.submit(new FactorialTask(5));

    System.out.println("Task submitted. Waiting for result...");
    Long result = future.get(); // Block and retrieve the result

    System.out.println("Factorial result: " + result);
    executor.shutdown();
  }
}
```

Output:

```
Task submitted. Waiting for result...
Factorial result: 120
```

Explanation:

- `Callable` returns a result (`Long` in this example).

- `Future` is used to retrieve the result of the asynchronous computation.
- The `get()` method blocks until the result is available.

6.1.4 Parallel Tasks and Performance with Fork/Join and Parallel Streams

6.1.4.0.1 Parallel Streams Parallel streams divide data into multiple threads for faster computation.

```
import java.util.stream.IntStream;

public class ParallelStreamExample {
  public static void main(String[] args) {
    System.out.println("Sum using Parallel Stream:");
    int sum = IntStream.range(1, 10)
    .parallel()
    .sum();
    System.out.println("Sum: " + sum);
  }
}
```

Output:

```
Sum using Parallel Stream:
Sum: 45
```

6.1.4.0.2 Best Practices for Parallel Tasks:

- Use thread pools for scalable and efficient task execution.
- Prefer `Callable` and `Future` for tasks that need to return results.
- Use parallel streams for CPU-bound operations on large datasets.
- Avoid modifying shared resources to prevent race conditions.
- Threads enable concurrent execution of tasks, improving program performance.
- Use the Executor Framework for managing thread pools efficiently.
- Use `Callable` and `Future` for asynchronous tasks that return results.
- Parallel streams simplify concurrent processing of large datasets.
- Always ensure thread safety when working with shared resources.

6.2 Synchronized Methods and Thread Safety

This chapter explains fundamental concepts and mechanisms of thread safety in Java concurrent programming. At its core, `thread safety` is essential for ensuring that multiple threads can safely access and modify shared resources without causing data corruption or unpredictable behavior. When multiple threads operate simultaneously, they can potentially create `race conditions` where the outcome depends on the precise timing of thread execution.

The `synchronized` keyword in Java provides a basic mechanism for thread synchronization, enabling exclusive execution where only one thread can execute a synchronized method or block at a time. This controlled access ensures that shared resources are protected, preventing concurrent modifications that could compromise data integrity. Through strategic use of synchronized methods and blocks, developers can implement thread-safe classes that maintain consistency in multithreaded environments.

Beyond basic synchronization, Java offers advanced tools for concurrent programming. `Locks` provide finer-grained control over thread synchronization, supporting complex scenarios like `read-write locks` where multiple readers can access data simultaneously while ensuring exclusive writer access. `Semaphores` manage access to limited resources, while `atomic variables` enable thread-safe operations without explicit locking mechanisms.

A critical challenge in concurrent programming is the prevention of `deadlocks`, which occur when two or more threads become permanently blocked, each waiting for resources held by others. Understanding deadlock scenarios is essential for developing robust multithreaded applications. Common prevention strategies include implementing consistent lock ordering, utilizing timeouts, and applying proper resource management techniques.

The selection of synchronization mechanisms depends heavily on specific requirements and performance trade-offs. While simple `synchronized` methods suffice for basic thread safety, complex scenarios often demand more sophisticated approaches using locks or atomic variables. Effective thread safety implementation requires not only correct usage of these tools but also a deep understanding of concurrent programming principles and potential pitfalls.

6.2.1 The `synchronized` Keyword

6.2.1.0.1 What is Thread Safety? Thread safety ensures that shared data is accessed safely when multiple threads are executing concurrently. Without thread

safety, race conditions occur, leading to unpredictable results.

6.2.1.0.2 Using the `synchronized` Keyword The `synchronized` keyword ensures that only one thread can access a critical section of code at a time. It can be applied to:

- Methods: Synchronizing an entire method.
- Blocks: Synchronizing a specific section of code.

6.2.1.0.3 Synchronized Method Example:

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

  // Synchronized method to increment the count
  public synchronized void increment() {
    count++;
  }

  public int getCount() {
    return count;
  }
}

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

    Runnable task = () -> {
      for (int i = 0; i < 1000; i++) {
        counter.increment();
      }
    };

    // Create two threads
    Thread t1 = new Thread(task);
    Thread t2 = new Thread(task);

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

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

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

Output:

```
Final Count: 2000
```


Explanation:

- The `synchronized` method ensures that only one thread can increment the count at a time.
- Without synchronization, multiple threads might update the count simultaneously, leading to incorrect results.

6.2.2 The `synchronized` Keyword

6.2.2.0.1 What is the `synchronized` Keyword? The `synchronized` keyword ensures that a block of code or a method is executed by only one thread at a time. Synchronization is essential when multiple threads access a shared resource.

6.2.2.0.2 Types of Synchronization

- Synchronized Method: Locks the entire method.
- Synchronized Block: Locks a specific section of code, improving granularity.

6.2.2.0.3 Example: Synchronized Method

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

  // Synchronized method
  public synchronized void increment() {
    count++;
  }

  public int getCount() {
    return count;
  }
}

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

    Runnable task = () -> {
      for (int i = 0; i < 1000; i++) {
        counter.increment();
      }
    };

    Thread t1 = new Thread(task);
    Thread t2 = new Thread(task);

    t1.start();
```



```
    t2.start();

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

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

Output:

```
Final Count: 2000
```

6.2.2.0.4 Synchronized Block A synchronized block allows locking only the critical section of the code, offering better performance.

```
class Counter {
  private int count = 0;
  private final Object lock = new Object();

  public void increment() {
    synchronized (lock) { // Lock only the critical section
      count++;
    }
  }

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

6.2.2.0.5 When to Use What

- Use synchronized methods when the entire method must be thread-safe.
- Use synchronized blocks for finer-grained locking to improve performance.

6.2.3 Locks, Semaphores, and Atomic Variables

6.2.3.0.1 Locks in Java Java's `Lock` interface (part of `java.util.concurrent.locks`) provides an advanced mechanism for thread synchronization. It allows more control over the locking process than `synchronized`.

6.2.3.0.2 Example: Using ReentrantLock


```java
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class SafeCounter {
  private int count = 0;
  private final Lock lock = new ReentrantLock();

  public void increment() {
    lock.lock(); // Acquire the lock
    try {
      count++;
    } finally {
      lock.unlock(); // Release the lock
    }
  }

  public int getCount() {
    return count;
  }
}

public class LockExample {
  public static void main(String[] args) throws InterruptedException {
    SafeCounter counter = new SafeCounter();

    Runnable task = () -> {
      for (int i = 0; i < 1000; i++) {
        counter.increment();
      }
    };

    Thread t1 = new Thread(task);
    Thread t2 = new Thread(task);

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

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

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

Output:

```
Final Count: 2000
```

6.2.3.0.3 Semaphores A semaphore is a synchronization tool that restricts access to a certain number of threads. For example, if only 3 threads are allowed access, other threads must wait.

```java
import java.util.concurrent.Semaphore;
```



```java
class Worker implements Runnable {
  private final Semaphore semaphore;

  public Worker(Semaphore semaphore) {
    this.semaphore = semaphore;
  }

  @Override
  public void run() {
    try {
      semaphore.acquire(); // Acquire a permit
      System.out.println(Thread.currentThread().getName() + " is working.");
      Thread.sleep(1000); // Simulate work
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      System.out.println(Thread.currentThread().getName() + " finished.");
      semaphore.release(); // Release the permit
    }
  }
}

public class SemaphoreExample {
  public static void main(String[] args) {
    Semaphore semaphore = new Semaphore(2); // Allow 2 threads at a time
    for (int i = 0; i < 5; i++) {
      new Thread(new Worker(semaphore)).start();
    }
  }
}
```

Output:

```
Thread-0 is working.
Thread-1 is working.
Thread-0 finished.
Thread-1 finished.
Thread-2 is working.
Thread-3 is working.
...
```

6.2.3.0.4 Atomic Variables

Atomic variables (e.g., `AtomicInteger`) allow thread-safe operations without explicit locking. They are part of the `java.util.concurrent.atomic` package.

6.2.3.0.5 Example: Using `AtomicInteger`

```java
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
```



```
  private AtomicInteger count = new AtomicInteger(0);

  public void increment() {
    count.incrementAndGet();
  }

  public int getCount() {
    return count.get();
  }
}

public class AtomicExample {
  public static void main(String[] args) throws InterruptedException {
    AtomicCounter counter = new AtomicCounter();

    Runnable task = () -> {
      for (int i = 0; i < 1000; i++) {
        counter.increment();
      }
    };

    Thread t1 = new Thread(task);
    Thread t2 = new Thread(task);

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

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

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

Output:

```
Final Count: 2000
```

6.2.4 Avoiding Deadlocks

6.2.4.0.1 What is a Deadlock? A deadlock occurs when two or more threads are waiting for each other's locks, leading to a situation where none can proceed.

6.2.4.0.2 Example of Deadlock:

```
class DeadlockExample {
  private final Object lock1 = new Object();
  private final Object lock2 = new Object();

  public void method1() {
    synchronized (lock1) {
      System.out.println("Lock1 acquired by " + Thread.currentThread().getName());
      synchronized (lock2) {
```



```
            System.out.println("Lock2 acquired by " + Thread.currentThread().getName());
        }
      }
    }

    public void method2() {
      synchronized (lock2) {
        System.out.println("Lock2 acquired by " + Thread.currentThread().getName());
        synchronized (lock1) {
          System.out.println("Lock1 acquired by " + Thread.currentThread().getName());
        }
      }
    }

    public static void main(String[] args) {
      DeadlockExample example = new DeadlockExample();

      Thread t1 = new Thread(example::method1);
      Thread t2 = new Thread(example::method2);

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

6.2.4.0.3 Avoiding Deadlocks:

- Always acquire locks in a consistent order.
- Use `tryLock()` from the `Lock` interface to avoid indefinite blocking.
- Use timeout mechanisms to detect and resolve deadlocks.

6.2.4.0.4 Summary of Thread Safety Mechanisms

- The `synchronized` keyword ensures that only one thread executes a critical section at a time.
- Locks (`ReentrantLock`) offer advanced control over synchronization.
- Semaphores manage access to shared resources for a limited number of threads.
- Atomic variables provide thread-safe operations without explicit locking.
- Deadlocks can be avoided by acquiring locks in a consistent order or using timeouts.

6.2.5 The `synchronized` Keyword

6.2.5.0.1 What is Synchronization? Synchronization ensures that only one thread can access a critical section of code or a shared resource at a time. In Java, synchronization is implemented using the `synchronized` keyword.

6.2.5.0.2 Synchronized Methods A synchronized method ensures that only one thread can execute the method at a time for a given object.

6.2.5.0.3 Example of Synchronized Methods

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

  // Synchronized method to ensure thread safety
  public synchronized void increment() {
    count++;
  }

  public int getCount() {
    return count;
  }
}

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

    Runnable task = () -> {
      for (int i = 0; i < 1000; i++) {
        counter.increment();
      }
    };

    // Create and start two threads
    Thread t1 = new Thread(task);
    Thread t2 = new Thread(task);

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

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

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

Output:

```
Final Count: 2000
```


6.2.5.0.4 Explanation:

- The `synchronized` keyword ensures that only one thread at a time can execute the `increment()` method.
- Without synchronization, race conditions would occur, leading to incorrect results.

6.2.6 Locks, Semaphores, and Atomic Variables

6.2.6.0.1 Locks in Java The `ReentrantLock` class in the `java.util.concurrent.locks` package provides an alternative to synchronized methods and blocks. It offers more flexibility, such as try-locking and interruptible locks.

6.2.6.0.2 Example: Using `ReentrantLock`

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class SafeCounter {
  private int count = 0;
  private final Lock lock = new ReentrantLock();

  public void increment() {
    lock.lock(); // Acquire the lock
    try {
      count++;
    } finally {
      lock.unlock(); // Release the lock
    }
  }

  public int getCount() {
    return count;
  }
}

public class LockExample {
  public static void main(String[] args) throws InterruptedException {
    SafeCounter counter = new SafeCounter();

    Runnable task = () -> {
      for (int i = 0; i < 1000; i++) {
        counter.increment();
      }
    };

    Thread t1 = new Thread(task);
    Thread t2 = new Thread(task);

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



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

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

Output:

```
Final Count: 2000
```

6.2.6.0.3 Semaphores for Controlling Access

A semaphore restricts the number of threads that can access a resource concurrently. It can be used for resource pooling or throttling.

6.2.6.0.4 Example: Using `Semaphore`

```
import java.util.concurrent.Semaphore;

class Resource {
  private final Semaphore semaphore = new Semaphore(2); // Allow 2 threads at a time

  public void useResource() {
    try {
      semaphore.acquire();
      System.out.println(Thread.currentThread().getName() + " is using the resource.");
      Thread.sleep(1000); // Simulate work
    } catch (InterruptedException e) {
      e.printStackTrace();
    } finally {
      System.out.println(Thread.currentThread().getName() + " has released the resource.");
      semaphore.release();
    }
  }
}

public class SemaphoreExample {
  public static void main(String[] args) {
    Resource resource = new Resource();

    Runnable task = resource::useResource;

    Thread t1 = new Thread(task, "Thread-1");
    Thread t2 = new Thread(task, "Thread-2");
    Thread t3 = new Thread(task, "Thread-3");
    Thread t4 = new Thread(task, "Thread-4");

    t1.start();
    t2.start();
    t3.start();
    t4.start();
  }
}
```


Output:

```
Thread-1 is using the resource.
Thread-2 is using the resource.
Thread-1 has released the resource.
Thread-2 has released the resource.
Thread-3 is using the resource.
Thread-4 is using the resource.
```

6.2.6.0.5 Atomic Variables The `java.util.concurrent.atomic` package provides atomic classes (e.g., `AtomicInteger`) for performing thread-safe operations without explicit locking.

6.2.6.0.6 Example: Using `AtomicInteger`

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
  private AtomicInteger count = new AtomicInteger(0);

  public void increment() {
    count.incrementAndGet();
  }

  public int getCount() {
    return count.get();
  }
}

public class AtomicVariableExample {
  public static void main(String[] args) throws InterruptedException {
    AtomicCounter counter = new AtomicCounter();

    Runnable task = () -> {
      for (int i = 0; i < 1000; i++) {
        counter.increment();
      }
    };

    Thread t1 = new Thread(task);
    Thread t2 = new Thread(task);

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

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

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


Output:

```
Final Count: 2000
```

6.2.7 Avoiding Deadlocks

6.2.7.0.1 What is a Deadlock? A deadlock occurs when two or more threads are waiting for each other to release resources, causing all threads to be blocked indefinitely.

6.2.7.0.2 Example of a Deadlock Scenario

```
class Resource {
  final Object lock1 = new Object();
  final Object lock2 = new Object();

  public void method1() {
    synchronized (lock1) {
      System.out.println(Thread.currentThread().getName() + " acquired lock1");
      synchronized (lock2) {
        System.out.println(Thread.currentThread().getName() + " acquired lock2");
      }
    }
  }

  public void method2() {
    synchronized (lock2) {
      System.out.println(Thread.currentThread().getName() + " acquired lock2");
      synchronized (lock1) {
        System.out.println(Thread.currentThread().getName() + " acquired lock1");
      }
    }
  }
}

public class DeadlockExample {
  public static void main(String[] args) {
    Resource resource = new Resource();

    Thread t1 = new Thread(() -> resource.method1(), "Thread-1");
    Thread t2 = new Thread(() -> resource.method2(), "Thread-2");

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

6.2.7.0.3 Avoiding Deadlocks: Best Practices

- Always acquire locks in a consistent order.

- Use try-lock mechanisms (e.g., `ReentrantLock.tryLock()`).
- Minimize the use of nested locks.
- Use higher-level concurrency tools like semaphores or atomic variables where possible.

6.2.7.0.4 Summary of Synchronized Methods and Thread Safety

- Use the `synchronized` keyword to ensure thread safety in critical sections.
- ReentrantLock provides greater flexibility over the `synchronized` keyword.
- Semaphores control access to limited resources.
- Atomic variables (`AtomicInteger`) simplify thread-safe operations without locking.
- Deadlocks can be avoided by consistent locking order and using try-lock mechanisms.

By mastering synchronization techniques, developers can write safe, efficient, and deadlock-free multithreaded applications in Java.

6.3 Project Loom and Virtual Threads

6.3.1 Background and Motivation

As of 2019, project Loom introduced a revolutionary approach to concurrent programming in Java through virtual threads, designed to solve the scalability challenges inherent in traditional platform threads. In contemporary server applications, especially those handling numerous I/O operations, the overhead of platform threads becomes a significant bottleneck. Each platform thread, typically consuming around 1MB of stack space and maintaining a one-to-one mapping with operating system threads, limits application scalability.

Consider the traditional approach:

```
// Traditional thread pooling approach
ExecutorService executor = Executors.newFixedThreadPool(100);
// Limited to 100 concurrent operations, regardless of hardware capacity
for (int i = 0; i < 10000; i++) {
  executor.submit(() -> {
    performIOOperation(); // Blocks an entire OS thread
  });
}
```


6.3.2 Virtual Threads: The Solution to Scale

Virtual threads provide a lightweight alternative that enables true scalability without compromising the familiar thread-per-request programming model. They are:

- Managed by the JDK rather than the operating system
- Extremely lightweight (approximately 200 bytes per thread)
- Automatically multiplexed onto platform threads (carrier threads)
- Ideal for I/O-bound operations

Example of modern approach using virtual threads:

```
try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
  // Can easily handle millions of concurrent tasks
  for (int i = 0; i < 1_000_000; i++) {
    executor.submit(() -> {
      performIOOperation(); // Automatically yields carrier thread
      return processResult();
    });
  }
}
```

6.3.3 Virtual Threads Architecture and Implementation

Virtual threads operate through a sophisticated mounting/unmounting mechanism:

1. A virtual thread is mounted on a carrier thread (platform thread) when it needs to execute code
2. When the virtual thread performs a blocking operation, it unmounts from the carrier
3. The carrier thread becomes available to mount another virtual thread
4. When the blocking operation completes, the virtual thread is scheduled to mount again

6.4 Advanced Concepts and Best Practices

6.4.1 Thread Scheduling and Management

The scheduling of virtual threads is handled by the JDK's ForkJoinPool:

```
Thread vThread = Thread.ofVirtual()
.name("custom-name")
.start(() -> {
  System.out.println("Current thread: " + Thread.currentThread());
  try {
    Thread.sleep(100); // Automatically unmounts
    System.out.println("After sleep: " + Thread.currentThread());
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
  }
});
```

6.4.2 Pinning and Performance Considerations

Virtual threads may become pinned to their carrier threads in certain scenarios:

- During synchronized blocks or methods
- When executing native methods or foreign functions

Example of potential pinning issue:

```
synchronized void performOperation() {
  // Virtual thread is pinned here
  networkOperation(); // Blocks the carrier thread unnecessarily
}

// Better approach using ReentrantLock
private final ReentrantLock lock = new ReentrantLock();
void performOperation() {
  lock.lock();
  try {
    networkOperation(); // Virtual thread can unmount
  } finally {
    lock.unlock();
  }
}
```

6.5 Memory Management and Resource Utilization

6.5.1 Stack Management

Virtual thread stacks are stored in the Java heap as stack chunk objects:

- Dynamic growth and shrinkage based on demand
- Efficient memory utilization
- Support for deep call stacks
- Garbage collection of unused stack chunks

6.5.2 Thread-Local Variables

While virtual threads support thread-local variables, their usage requires careful consideration:

```
ThreadLocal<String> threadLocal = new ThreadLocal<>();
Thread vThread = Thread.ofVirtual().start(() -> {
  threadLocal.set("context");
  try {
    performOperation();
  } finally {
    threadLocal.remove(); // Clean up to prevent memory leaks
  }
});
```

6.6 Practical Applications and Usage Patterns

6.6.1 HTTP Server Example

With an example we want to demonstrate a quick snippet implementation of a scalable HTTP server using Java's virtual threads and structured concurrency:

```
class HttpServer {
  void start() throws IOException {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
      ServerSocket server = new ServerSocket(8080);
      while (true) {
        Socket socket = server.accept();
        scope.fork(() -> handleRequest(socket));
      }
    }
  }

  private void handleRequest(Socket socket) throws IOException {
    try (socket) {
      // Process HTTP request
      // Virtual thread automatically unmounts during I/O
      processRequest(socket.getInputStream());
      sendResponse(socket.getOutputStream());
    }
  }
}
```


The design achieves high scalability through:

- Creating a dedicated virtual thread per connection, avoiding thread pool limitations
- Automatic resource management via structured concurrency
- Efficient I/O handling through virtual thread unmounting
- Simple, synchronous-style code that's easy to maintain

This approach contrasts with traditional server implementations that typically require either:

- Complex thread pooling with limited concurrency
- Complicated asynchronous programming patterns
- Resource-heavy platform threads

6.6.2 Database Operations Example

Here is another snippet for efficient database access pattern using virtual threads:

```
class DatabaseService {
  CompletableFuture<List<Result>> performParallelQueries(
  List<Query> queries) {
    try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
      return CompletableFuture.supplyAsync(() ->
      queries.stream()
      .parallel()
      .map(this::executeQuery)
      .collect(Collectors.toList()),
      executor
      );
    }
  }

  private Result executeQuery(Query query) {
    // Each query runs in its own virtual thread
    // Automatically yields during database I/O
    return databaseConnection.execute(query);
  }
}
```


6.7 Summary of Virtual Threads

Virtual threads represent a significant advancement in Java concurrency, enabling developers to write highly scalable applications while maintaining the simplicity of the thread-per-request model. By understanding and properly implementing virtual threads, developers can achieve optimal resource utilization and improved application performance without sacrificing code readability or maintainability.

Chapter 7

Networking in Java

7.1 Java Networking: Sockets and URL Connections

Java provides comprehensive support for networking, allowing developers to build applications that communicate over the network. This chapter focuses on:

- TCP and UDP sockets for connection-oriented and connectionless communication.
- HTTP communication using URL connections.

7.1.1 TCP and UDP Sockets

7.1.1.0.1 What are Sockets? A Socket is an endpoint for communication between two machines. Java supports:

- TCP Sockets: Reliable, connection-oriented communication.
- UDP Sockets: Unreliable, connectionless communication.

7.1.1.0.2 TCP Socket Programming TCP (Transmission Control Protocol) ensures reliable communication. It is used in applications like web browsing and email.

7.1.1.0.3 Example: TCP Server and Client TCP Server:

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

public class TCPServer {
  public static void main(String[] args) {
    try (ServerSocket serverSocket = new ServerSocket(12345)) {
      System.out.println("Server is listening on port 12345");

      while (true) {
        Socket socket = serverSocket.accept();
        System.out.println("New client connected");

        BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        PrintWriter writer = new PrintWriter(socket.getOutputStream(), true);

        String message = reader.readLine();
        System.out.println("Received: " + message);

        writer.println("Echo: " + message);
        socket.close();
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```


TCP Client:

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

public class TCPClient {
  public static void main(String[] args) {
    try (Socket socket = new Socket("localhost", 12345)) {
      PrintWriter writer = new PrintWriter(socket.getOutputStream(), true);
      BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));

      writer.println("Hello, Server!");
      String response = reader.readLine();
      System.out.println("Server response: " + response);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

Output: **Server:**

```
Server is listening on port 12345
New client connected
Received: Hello, Server!
```

Client:

```
Server response: Echo: Hello, Server!
```

7.1.1.0.4 UDP Socket Programming

UDP (User Datagram Protocol) is faster but less reliable. It is used in real-time applications like video streaming and gaming.

7.1.1.0.5 Example: UDP Server and Client

UDP Server:

```
import java.net.*;

public class UDPServer {
  public static void main(String[] args) {
    try (DatagramSocket socket = new DatagramSocket(12345)) {
      System.out.println("UDP Server is running on port 12345");

      byte[] buffer = new byte[1024];
      DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

      socket.receive(packet);
      String message = new String(packet.getData(), 0, packet.getLength());
      System.out.println("Received: " + message);

      String response = "Echo: " + message;
      byte[] responseData = response.getBytes();
      DatagramPacket responsePacket = new DatagramPacket(responseData, responseData.length, packet.getAddress(), packet.getPort
```



```
      socket.send(responsePacket);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

UDP Client:

```
import java.net.*;

public class UDPClient {
  public static void main(String[] args) {
    try (DatagramSocket socket = new DatagramSocket()) {
      String message = "Hello, UDP Server!";
      byte[] buffer = message.getBytes();
      InetAddress address = InetAddress.getByName("localhost");

      DatagramPacket packet = new DatagramPacket(buffer, buffer.length, address, 12345);
      socket.send(packet);

      byte[] responseBuffer = new byte[1024];
      DatagramPacket responsePacket = new DatagramPacket(responseBuffer, responseBuffer.length);
      socket.receive(responsePacket);

      String response = new String(responsePacket.getData(), 0, responsePacket.getLength());
      System.out.println("Server response: " + response);
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Output: **Server:**

```
UDP Server is running on port 12345
Received: Hello, UDP Server!
```

Client:

```
Server response: Echo: Hello, UDP Server!
```

7.1.2 HTTP Communication Using URL Connections

7.1.2.0.1 What is HTTP Communication? HTTP (HyperText Transfer Protocol) is the foundation of data communication on the web. Java provides the `HttpURLConnection` class for sending and receiving HTTP requests and responses.

7.1.2.0.2 Example: Sending an HTTP GET Request

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

public class HttpGetExample {
  public static void main(String[] args) {
    try {
      URL url = new URL("https://jsonplaceholder.typicode.com/posts/1");
      HttpURLConnection connection = (HttpURLConnection) url.openConnection();
      connection.setRequestMethod("GET");

      int responseCode = connection.getResponseCode();
      System.out.println("Response Code: " + responseCode);

      BufferedReader reader = new BufferedReader(new InputStreamReader(connection.getInputStream()));
      String line;
      while ((line = reader.readLine()) != null) {
        System.out.println(line);
      }
      reader.close();
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

Output: The application prints the JSON response from the specified URL:

```
Response Code: 200
{
        "userId": 1,
        "id": 1,
        "title": "sample title",
        "body": "sample body text"
}
```

7.1.2.0.3 Example: Sending an HTTP POST Request

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

public class HttpPostExample {
  public static void main(String[] args) {
    try {
      URL url = new URL("https://jsonplaceholder.typicode.com/posts");
      HttpURLConnection connection = (HttpURLConnection) url.openConnection();
      connection.setRequestMethod("POST");
      connection.setDoOutput(true);
      connection.setRequestProperty("Content-Type", "application/json");

      String jsonInputString = "{\"title\":\"foo\",\"body\":\"bar\",\"userId\":1}";
```



```
      try (OutputStream os = connection.getOutputStream()) {
        byte[] input = jsonInputString.getBytes("utf-8");
        os.write(input, 0, input.length);
      }

      int responseCode = connection.getResponseCode();
      System.out.println("Response Code: " + responseCode);

      BufferedReader reader = new BufferedReader(new InputStreamReader(connection.getInputStream(), "utf-8"));
      StringBuilder response = new StringBuilder();
      String line;
      while ((line = reader.readLine()) != null) {
        response.append(line.trim());
      }
      System.out.println("Response: " + response.toString());
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

Output:

```
Response Code: 201
Response: {"id":101}
```

7.1.3 Best Practices for Java Networking

7.1.3.0.1 Best Practices:

- Always close sockets and streams to free resources.
- Use timeouts to handle unresponsive servers (`setSoTimeout` for sockets).
- Prefer higher-level HTTP clients (e.g., Apache HttpClient or Java's `HttpClient` introduced in Java 11) for advanced features.
- Validate and sanitize input data in networking applications to prevent injection attacks.
- TCP Sockets: Used for reliable, connection-oriented communication.
- UDP Sockets: Used for fast, connectionless communication.
- HTTP Communication: Enables web-based communication using GET and POST methods.

By mastering Java's networking features, developers can build robust, efficient, and secure networked applications.

Chapter 8

Basic Graphical User Interfaces in Java

8.1 GUI Programming with Swing

Swing is a part of Java's standard library used to create Graphical User Interfaces (GUIs). It provides a set of lightweight components and a flexible architecture for building desktop applications. This chapter explains key Swing components, layout managers, and event listeners.

8.1.1 Introduction to Swing

8.1.1.0.1 What is Swing? Swing is part of the Java Foundation Classes (JFC) and provides:

- Lightweight components (not platform-dependent).
- MVC (Model-View-Controller) architecture.
- A variety of pre-built components such as `JFrame`, `JPanel`, `JButton`, etc.

8.1.1.0.2 Basic Swing Program Structure A basic Swing program consists of:

- JFrame: The main application window.
- JPanel: A container for organizing components.
- JButton, JLabel, JTextField, etc.: GUI components.
- Event Listeners: To handle user actions (e.g., button clicks).

8.1.2 Swing Components: `JFrame`, `JPanel`, Buttons

8.1.2.0.1 Example: Basic Swing Application with a Button

```
import javax.swing.*;

public class BasicSwingApp {
  public static void main(String[] args) {
    // Schedule GUI creation on Event Dispatch Thread
    SwingUtilities.invokeLater(() -> {
      // Create a JFrame
      JFrame frame = new JFrame("Basic Swing Example");
      frame.setSize(400, 200);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create a JPanel
      JPanel panel = new JPanel();
```



```
        // Add a JButton to the panel
        JButton button = new JButton("Click Me!");
        panel.add(button);

        // Add a JLabel to display a message
        JLabel label = new JLabel("Welcome to Swing GUI!");
        panel.add(label);

        // Add panel to the frame
        frame.add(panel);

        // Make the frame visible
        frame.setVisible(true);
    });
  }
}
```

8.1.2.0.2 Explanation:

- `JFrame` creates the main window.
- `JPanel` organizes components.
- `JButton` and `JLabel` are added to the panel.
- `SwingUtilities.invokeLater()` ensures thread-safe creation of the GUI on the Event Dispatch Thread (EDT).

Output: The application opens a window titled "Basic Swing Example" with a button labeled "Click Me!" and a welcome message.

8.1.3 Layout Managers in Swing

8.1.3.0.1 What are Layout Managers? Layout managers define how components are arranged in a container. Common layout managers include:

- **`FlowLayout`**: Components are arranged in a flow, left-to-right.
- **`BorderLayout`**: Divides the container into North, South, East, West, and Center.
- **`GridLayout`**: Arranges components in a grid (rows and columns).
- **`BoxLayout`**: Places components in a single row or column.

8.1.3.0.2 Example: Using `FlowLayout` and `GridLayout`

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

public class LayoutExample {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
      JFrame frame = new JFrame("Layout Manager Example");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setSize(300, 200);

      // FlowLayout Example
      JPanel flowPanel = new JPanel(new FlowLayout());
      flowPanel.add(new JButton("Button 1"));
      flowPanel.add(new JButton("Button 2"));

      // GridLayout Example
      JPanel gridPanel = new JPanel(new GridLayout(2, 2));
      gridPanel.add(new JButton("Grid 1"));
      gridPanel.add(new JButton("Grid 2"));
      gridPanel.add(new JButton("Grid 3"));
      gridPanel.add(new JButton("Grid 4"));

      // Add panels to the frame
      frame.setLayout(new BorderLayout());
      frame.add(flowPanel, BorderLayout.NORTH);
      frame.add(gridPanel, BorderLayout.CENTER);

      frame.setVisible(true);
    });
  }
}
```

Output: The application window has buttons arranged in a flow at the top and a 2x2 grid in the center.

8.1.4 Event Listeners in Swing

8.1.4.0.1 What are Event Listeners? Event listeners handle user actions such as button clicks, key presses, or mouse movements. The listener interfaces include:

- `ActionListener`: Handles button clicks.
- `MouseListener`: Handles mouse events.
- `KeyListener`: Handles key events.

8.1.4.0.2 Example: Handling Button Click Events


```java
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class EventListenerExample {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
      JFrame frame = new JFrame("Event Listener Example");
      frame.setSize(300, 150);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      JPanel panel = new JPanel();

      // Create a button
      JButton button = new JButton("Click Me!");
      JLabel label = new JLabel("Waiting for action...");

      // Add ActionListener to the button
      button.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
          label.setText("Button was clicked!");
        }
      });

      // Add components to the panel
      panel.add(button);
      panel.add(label);

      // Add panel to the frame
      frame.add(panel);
      frame.setVisible(true);
    });
  }
}
```

Output: The GUI displays a button and a label. When the button is clicked, the label updates to show: ”Button was clicked!”.

8.1.4.0.3 Using Lambda Expressions for Event Listeners (Java 8+):

```java
button.addActionListener(e -> label.setText("Button was clicked!"));
```

Using lambda expressions simplifies event listener code for functional interfaces like `ActionListener`.

8.1.5 Best Practices for GUI Programming in Swing

8.1.5.0.1 Best Practices:

- Always update the GUI components on the Event Dispatch Thread (EDT) using `SwingUtilities.invokeLater()`.

- Use layout managers to ensure consistent component arrangement across different screen sizes.
- Avoid blocking the EDT for long-running tasks; use worker threads (e.g., `SwingWorker`).
- Use lambda expressions for cleaner event listener code.
- Keep business logic separate from GUI code to improve maintainability.

8.1.5.0.2 Summary of GUI Programming with Swing

- Swing provides lightweight and flexible components such as `JFrame`, `JPanel`, and `JButton`.
- Layout managers like `FlowLayout`, `BorderLayout`, and `GridLayout` organize GUI components effectively.
- Event listeners (e.g., `ActionListener`) handle user interactions like button clicks.
- Use `SwingUtilities.invokeLater()` to ensure thread-safe GUI updates.

By understanding Swing components, layout managers, and event handling, developers can create interactive and well-structured desktop applications in Java.

8.2 Event Handling in GUI Applications

Event handling in GUI applications refers to responding to user interactions such as button clicks, mouse movements, and keyboard input. In Java Swing, events are processed using the Event Delegation Model, where event listeners handle and manage events.

This chapter explores key event listeners like `ActionListener`, `MouseListener`, and `KeyListener` along with the Event Delegation Model.

8.2.1 Event Delegation Model

8.2.1.0.1 What is the Event Delegation Model? The Event Delegation Model is a mechanism in Java where:

- An event source generates events (e.g., a button click).
- Event listeners listen for specific events.
- Listeners handle the events when they occur.

8.2.1.0.2 Key Components of the Event Delegation Model:

- Event Source: The component that generates an event (e.g., `JButton`, `JTextField`).
- Event Listener: An interface that listens for events and defines methods to handle them.
- Event Object: Encapsulates details about the event (e.g., `ActionEvent`, `MouseEvent`).

8.2.2 ActionListener: Handling Button Click Events

8.2.2.0.1 What is `ActionListener`? The `ActionListener` interface is used to handle action events like button clicks or menu item selections. It contains the `actionPerformed()` method.

8.2.2.0.2 Example: Handling Button Click Events

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

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



```
    // Create a JFrame
    JFrame frame = new JFrame("ActionListener Example");
    frame.setSize(300, 150);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Create a JButton
    JButton button = new JButton("Click Me!");

    // Add an ActionListener
    button.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(frame, "Button was clicked!");
      }
    });

    // Add button to the frame
    frame.add(button);
    frame.setVisible(true);
  }
}
```

Output: A window with a button labeled "Click Me!" appears. When the button is clicked, a dialog box displays: "Button was clicked!".

8.2.2.0.3 Using Lambda Expressions with `ActionListener` (Java 8+):

```
button.addActionListener(e -> JOptionPane.showMessageDialog(frame, "Button was clicked!"));
```

8.2.3 MouseListener: Handling Mouse Events

8.2.3.0.1 What is `MouseListener`? The `MouseListener` interface handles mouse events like clicks, movement, entering, or exiting a component.

8.2.3.0.2 Example: Handling Mouse Events

```
import javax.swing.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

public class MouseListenerExample {
  public static void main(String[] args) {
    JFrame frame = new JFrame("MouseListener Example");
    frame.setSize(300, 200);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JLabel label = new JLabel("Hover or Click on this label!");
    label.setHorizontalAlignment(SwingConstants.CENTER);

    // Add MouseListener
    label.addMouseListener(new MouseListener() {
      @Override
```



```
      public void mouseClicked(MouseEvent e) {
        System.out.println("Mouse clicked at: " + e.getX() + ", " + e.getY());
      }

      @Override
      public void mouseEntered(MouseEvent e) {
        label.setText("Mouse entered!");
      }

      @Override
      public void mouseExited(MouseEvent e) {
        label.setText("Mouse exited!");
      }

      @Override
      public void mousePressed(MouseEvent e) {
        System.out.println("Mouse pressed.");
      }

      @Override
      public void mouseReleased(MouseEvent e) {
        System.out.println("Mouse released.");
      }
    });

    frame.add(label);
    frame.setVisible(true);
  }
}
```

Output:

- "Mouse entered!" is displayed when the mouse hovers over the label.
- "Mouse exited!" is displayed when the mouse leaves the label.
- Console logs the click coordinates and other events.

8.2.4 KeyListener: Handling Keyboard Events

8.2.4.0.1 What is `KeyListener`? The `KeyListener` interface handles keyboard events such as key presses, releases, and typing.

8.2.4.0.2 Example: Handling Keyboard Events

```
import javax.swing.*;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

public class KeyListenerExample {
  public static void main(String[] args) {
    JFrame frame = new JFrame("KeyListener Example");
```



```
    frame.setSize(300, 150);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JTextField textField = new JTextField(20);
    JLabel label = new JLabel("Type something...");

    // Add KeyListener to the text field
    textField.addKeyListener(new KeyListener() {
      @Override
      public void keyTyped(KeyEvent e) {
        label.setText("Key Typed: " + e.getKeyChar());
      }

      @Override
      public void keyPressed(KeyEvent e) {
        System.out.println("Key Pressed: " + e.getKeyCode());
      }

      @Override
      public void keyReleased(KeyEvent e) {
        System.out.println("Key Released: " + e.getKeyCode());
      }
    });

    JPanel panel = new JPanel();
    panel.add(textField);
    panel.add(label);

    frame.add(panel);
    frame.setVisible(true);
  }
}
```

Output:

- Typing in the text field updates the label with the character typed.
- Console logs the key codes for key presses and releases.

8.2.5 Summary of Event Listeners

8.2.5.0.1 Summary of Event Handling Java Swing provides various event listeners to handle different types of user interactions:

- `ActionListener`: Handles button clicks and action events.
- `MouseListener`: Handles mouse interactions like clicks, entering, and exiting components.
- `KeyListener`: Handles keyboard input events such as key presses and releases.

8.2.5.0.2 Event Delegation Model

The Event Delegation Model ensures clean separation of event generation and handling:

- The Event Source generates events.
- The Event Listener processes events using specific callback methods.
- Event listeners are registered with components using methods like `addActionListener()` or `addMouseListener()`.

8.2.5.0.3 Best Practices for Event Handling

- Always perform time-consuming tasks on separate threads, not on the Event Dispatch Thread (EDT).
- Use lambda expressions (Java 8+) for cleaner event listener code.
- Avoid writing all event logic directly in the listener; delegate to helper methods for readability.
- Test components thoroughly to ensure that event listeners respond correctly.

8.2.5.0.4 Summary of GUI Event Handling in Java

- Swing provides event listeners to handle user interactions.
- Use `ActionListener` for action events like button clicks.
- Use `MouseListener` for mouse events and `KeyListener` for keyboard input.
- The Event Delegation Model separates event generation (source) from event handling (listener).

By mastering event handling, developers can create interactive and responsive Java Swing applications with ease.

8.3 Implementing Graphical User Interfaces (GUIs)

Building interactive and user-friendly Graphical User Interfaces (GUIs) is an essential skill for Java developers. Using Swing, developers can design multi-panel applications, organize components logically, and follow best practices to ensure maintainable and responsive designs.

This chapter focuses on implementing multi-panel GUI applications and explores best practices for GUI design.

8.3.1 Building Multi-Panel GUI Applications

8.3.1.0.1 What is a Multi-Panel GUI? A multi-panel GUI consists of multiple panels organized within a main window. Panels can be used to group components logically and provide a clean user interface.

8.3.1.0.2 Swing Containers for Multi-Panel Applications Key Swing containers used for multi-panel applications include:

- `JPanel`: A container to hold and group components.
- `JFrame`: The main application window.
- `JSplitPane`: Divides a window into two resizable areas.
- `JTabbedPane`: Provides tabbed navigation between multiple panels.

8.3.1.0.3 Example: Multi-Panel GUI Application with Navigation Tabs

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

public class MultiPanelGUI {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
      // Create the main JFrame
      JFrame frame = new JFrame("Multi-Panel GUI Application");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setSize(400, 300);

      // Create a JTabbedPane for panel navigation
      JTabbedPane tabbedPane = new JTabbedPane();

      // Panel 1: Welcome Panel
      JPanel panel1 = new JPanel();
      panel1.add(new JLabel("Welcome to the Application!"));
      panel1.add(new JButton("Click Me"));
```



```
        // Panel 2: User Information
        JPanel panel2 = new JPanel(new GridLayout(2, 2));
        panel2.add(new JLabel("Name:"));
        panel2.add(new JTextField(15));
        panel2.add(new JLabel("Email:"));
        panel2.add(new JTextField(15));

        // Panel 3: About
        JPanel panel3 = new JPanel();
        panel3.add(new JLabel("About the Application:"));
        JTextArea aboutText = new JTextArea(5, 20);
        aboutText.setText("This is a demo multi-panel GUI application.");
        aboutText.setEditable(false);
        panel3.add(new JScrollPane(aboutText));

        // Add panels to the tabbed pane
        tabbedPane.addTab("Welcome", panel1);
        tabbedPane.addTab("User Info", panel2);
        tabbedPane.addTab("About", panel3);

        // Add the tabbed pane to the frame
        frame.add(tabbedPane, BorderLayout.CENTER);
        frame.setVisible(true);
    });
  }
}
```

Explanation:

- A `JTabbedPane` organizes panels into tabs for easy navigation.
- Panels are created with different layouts (`FlowLayout`, `GridLayout`).
- Components like `JLabel`, `JButton`, `JTextField`, and `JTextArea` are added to panels.

Output: The GUI displays three tabs:

- Welcome: Contains a label and a button.
- User Info: Contains fields for name and email input.
- About: Displays an informational text area.

8.3.2 Combining Panels with Layout Managers

8.3.2.0.1 Nesting Panels for Complex Layouts For complex GUIs, panels can be nested within other panels, and different layout managers can be used simultaneously.

8.3.2.0.2 Example: Combining Panels with Different Layouts

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

public class NestedPanelExample {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
      JFrame frame = new JFrame("Nested Panels Example");
      frame.setSize(400, 300);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Main panel with BorderLayout
      JPanel mainPanel = new JPanel(new BorderLayout());

      // Top panel with FlowLayout
      JPanel topPanel = new JPanel(new FlowLayout());
      topPanel.add(new JLabel("Top Panel"));
      topPanel.add(new JButton("Button 1"));

      // Center panel with GridLayout
      JPanel centerPanel = new JPanel(new GridLayout(2, 2));
      centerPanel.add(new JButton("1"));
      centerPanel.add(new JButton("2"));
      centerPanel.add(new JButton("3"));
      centerPanel.add(new JButton("4"));

      // Bottom panel with FlowLayout
      JPanel bottomPanel = new JPanel(new FlowLayout());
      bottomPanel.add(new JLabel("Bottom Panel"));
      bottomPanel.add(new JButton("Button 2"));

      // Combine panels
      mainPanel.add(topPanel, BorderLayout.NORTH);
      mainPanel.add(centerPanel, BorderLayout.CENTER);
      mainPanel.add(bottomPanel, BorderLayout.SOUTH);

      frame.add(mainPanel);
      frame.setVisible(true);
    });
  }
}
```

Output: The application window contains:

- A top panel with a label and a button (FlowLayout).
- A center panel with a 2x2 grid of buttons (GridLayout).
- A bottom panel with another label and button (FlowLayout).

8.3.2.0.3 Example: Setting Look and Feel and Adding Tooltips

```
import javax.swing.*;
```



```
public class BestPracticesExample {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
      try {
        // Set cross-platform look and feel
        UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName());
      } catch (Exception e) {
        e.printStackTrace();
      }

      // Create JFrame
      JFrame frame = new JFrame("Best Practices Example");
      frame.setSize(300, 150);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create a button with a tooltip
      JButton button = new JButton("Hover Me!");
      button.setToolTipText("Click this button to perform an action.");

      // Add button to the frame
      frame.add(button);
      frame.setVisible(true);
    });
  }
}
```

Output: The GUI displays a button with a tooltip that appears when the user hovers over it.

8.3.3 GUI Implementation and Design Best Practices

- Use Layout Managers: Avoid hardcoding component positions; use layout managers like `BorderLayout`, `GridLayout`, and `FlowLayout`.
- Organize Components with Panels: Group related components using `JPanel`.
- Consistent Look and Feel: Use `UIManager` to set the application look and feel.
- Thread Safety: Always update the GUI on the Event Dispatch Thread (EDT) using `SwingUtilities.invokeLater()`.
- Responsive UI: Use background threads (e.g., `SwingWorker`) for long-running tasks to keep the UI responsive.
- Use Icons and Tooltips: Add icons and tooltips for better user experience.
- Multi-Panel Applications: Use `JPanel`, `JTabbedPane`, and layout managers to organize components effectively.

- Nesting Panels: Combine panels with different layouts for complex GUIs.
- Look and Feel: Use `UIManager` to ensure a consistent user interface across platforms.
- Thread Safety: Always update the GUI on the Event Dispatch Thread (EDT).
- Responsive UIs: Use background threads (`SwingWorker`) for time-consuming tasks.

Chapter 9

Famous Design Patterns in Java

9.1 Understanding Design Patterns and Their Implementati

9.1.1 Introduction to Design Patterns

Design patterns represent battle-tested solutions to common software design challenges. They provide templates for building maintainable, scalable, and robust applications while promoting code reuse and reducing common errors. This comprehensive chapter explores the theory and practical implementation of essential design patterns.

9.1.1.1 Creational Patterns

Patterns focusing on object creation mechanisms:

- **Singleton Pattern**
  - Ensures single instance creation
  - Controls global state
  - Common in configuration management
- **Factory Patterns Family**
  - Factory Method
  - Abstract Factory
  - Simple Factory
  - Static Factory
- **Builder Pattern**
  - Complex object construction
  - Fluent interfaces
  - Parameter validation
- **Object Creation Patterns**
  - Prototype
  - Object Pool
  - Immutable Object

9.1.1.2 Structural Patterns

Patterns establishing object relationships:

- **Composition Patterns**
  - Composite
  - Bridge
  - Facade
  - Adapter
- **Decorator Pattern**
  - Dynamic behavior addition
  - Runtime flexibility
  - Stream implementations
- **Proxy Pattern**
  - Access control
  - Lazy loading
  - Remote resource management

9.1.1.3 Behavioral Patterns

Patterns managing object communication:

- **Observer Pattern**
  - Event handling
  - Loose coupling
  - State change notification
- **Strategy Pattern**
  - Algorithm encapsulation
  - Runtime behavior switching
  - Policy management

- **Command Pattern**
  - Action encapsulation
  - Undo/Redo support
  - Queue management

9.1.2 Architectural Patterns

Higher-level organizational patterns:

- **MVC Pattern**
  - Separation of concerns
  - UI management
  - Data presentation
- **Repository Pattern**
  - Data access abstraction
  - CRUD operations
  - Query optimization
- **Service Layer Pattern**
  - Business logic organization
  - Application service definition
  - Transaction management

9.1.3 Enterprise Integration Patterns

Patterns for large-scale systems:

- **Dependency Injection**
  - Inversion of Control
  - Component management
  - Testing support
- **Service Locator**

  - Service discovery
  - Runtime binding
  - Resource management
- **Event Aggregator**
  - Event management
  - Message routing
  - Pub/Sub implementation

9.1.4 Implementation Best Practices

- **Pattern Selection**
  - Requirements analysis
  - Performance considerations
  - Maintenance implications
- **Code Quality**
  - SOLID principles
  - Clean code practices
  - Documentation standards
- **Testing Strategies**
  - Unit testing approaches
  - Integration testing
  - Pattern-specific tests

9.1.5 Common Anti-Patterns and Pitfalls

- **Design Issues**
  - Over-engineering
  - Premature optimization
  - Pattern misuse

- **Implementation Problems**
  - Tight coupling
  - God objects
  - Inappropriate inheritance
- **Maintenance Challenges**
  - Code rigidity
  - Poor documentation
  - Technical debt

9.1.6 Creational Design Patterns

9.1.6.0.1 Singleton Pattern Ensures a class has only one instance and provides a global access point to it.

```
class Singleton {
  private static Singleton instance;

  private Singleton() {} // Private constructor

  public static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}

public class SingletonExample {
  public static void main(String[] args) {
    Singleton s1 = Singleton.getInstance();
    Singleton s2 = Singleton.getInstance();
    System.out.println(s1 == s2); // true
  }
}
```

9.1.6.0.2 Factory Method Pattern Defines an interface for creating objects, but allows subclasses to decide which class to instantiate.

```
abstract class Product {
  abstract void use();
}

class ConcreteProductA extends Product {
  void use() {
```



```
    System.out.println("Using Product A");
  }
}

class ConcreteProductB extends Product {
  void use() {
    System.out.println("Using Product B");
  }
}

abstract class Factory {
  abstract Product createProduct();
}

class FactoryA extends Factory {
  Product createProduct() {
    return new ConcreteProductA();
  }
}

class FactoryB extends Factory {
  Product createProduct() {
    return new ConcreteProductB();
  }
}

public class FactoryMethodExample {
  public static void main(String[] args) {
    Factory factory = new FactoryA();
    Product product = factory.createProduct();
    product.use();
  }
}
```

9.1.6.0.3 Builder Pattern

Separates the construction of a complex object from its representation.

```
class Car {
  private String engine;
  private int wheels;

  static class Builder {
    private String engine;
    private int wheels;

    Builder setEngine(String engine) {
      this.engine = engine;
      return this;
    }

    Builder setWheels(int wheels) {
      this.wheels = wheels;
      return this;
    }

    Car build() {
```

```
      Car car = new Car();
      car.engine = this.engine;
      car.wheels = this.wheels;
      return car;
    }
  }

  @Override
  public String toString() {
    return "Car [engine=" + engine + ", wheels=" + wheels + "]";
  }
}

public class BuilderExample {
  public static void main(String[] args) {
    Car car = new Car.Builder().setEngine("V8").setWheels(4).build();
    System.out.println(car);
  }
}
```

9.1.6.0.4 Prototype Pattern

Creates a new object by copying an existing object.

```
class Prototype implements Cloneable {
  String name;

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

  @Override
  protected Prototype clone() throws CloneNotSupportedException {
    return (Prototype) super.clone();
  }
}

public class PrototypeExample {
  public static void main(String[] args) throws CloneNotSupportedException {
    Prototype original = new Prototype("Original");
    Prototype copy = original.clone();

    System.out.println("Original: " + original.name);
    System.out.println("Copy: " + copy.name);
  }
}
```

9.1.6.0.5 Object Pool Pattern

Manages a pool of reusable objects.

```
import java.util.Queue;
import java.util.LinkedList;

class ObjectPool {
  private Queue<String> pool = new LinkedList<>();
```



```
  public ObjectPool() {
    for (int i = 0; i < 5; i++) {
      pool.add("Resource-" + i);
    }
  }

  public String acquire() {
    return pool.poll();
  }

  public void release(String resource) {
    pool.add(resource);
  }
}

public class ObjectPoolExample {
  public static void main(String[] args) {
    ObjectPool pool = new ObjectPool();

    String resource = pool.acquire();
    System.out.println("Acquired: " + resource);

    pool.release(resource);
    System.out.println("Released: " + resource);
  }
}
```

9.1.7 Structural Design Patterns

9.1.7.0.1 Adapter Pattern

Allows incompatible interfaces to work together.

```
interface Target {
  void request();
}

class Adaptee {
  void specificRequest() {
    System.out.println("Adaptee's specific request");
  }
}

class Adapter implements Target {
  private Adaptee adaptee;

  Adapter(Adaptee adaptee) {
    this.adaptee = adaptee;
  }

  @Override
  public void request() {
    adaptee.specificRequest();
  }
}

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



```
    Adaptee adaptee = new Adaptee();
    Target adapter = new Adapter(adaptee);
    adapter.request();
  }
}
```

9.1.7.0.2 Decorator Pattern

Adds functionality to an object dynamically.

```
interface Component {
  void operation();
}

class ConcreteComponent implements Component {
  public void operation() {
    System.out.println("Base operation");
  }
}

class Decorator implements Component {
  private Component component;

  Decorator(Component component) {
    this.component = component;
  }

  @Override
  public void operation() {
    component.operation();
    System.out.println("Added functionality");
  }
}

public class DecoratorExample {
  public static void main(String[] args) {
    Component component = new ConcreteComponent();
    Component decorated = new Decorator(component);

    decorated.operation();
  }
}
```

9.1.7.0.3 Facade Pattern

Provides a simplified interface to a larger body of code.

```
class SubsystemA {
  void operationA() {
    System.out.println("Subsystem A operation");
  }
}

class SubsystemB {
  void operationB() {
    System.out.println("Subsystem B operation");
  }
```



```
}

class Facade {
  private SubsystemA subsystemA = new SubsystemA();
  private SubsystemB subsystemB = new SubsystemB();

  void operation() {
    subsystemA.operationA();
    subsystemB.operationB();
  }
}

public class FacadeExample {
  public static void main(String[] args) {
    Facade facade = new Facade();
    facade.operation();
  }
}
```

Output:

```
Subsystem A operation
Subsystem B operation
```

9.1.7.0.4 Mediator Pattern Defines an object that encapsulates how other objects interact.

```
interface Mediator {
  void notify(String message, Colleague colleague);
}

class ConcreteMediator implements Mediator {
  private ColleagueA colleagueA;
  private ColleagueB colleagueB;

  void setColleagues(ColleagueA a, ColleagueB b) {
    this.colleagueA = a;
    this.colleagueB = b;
  }

  public void notify(String message, Colleague colleague) {
    if (colleague == colleagueA) {
      colleagueB.receive(message);
    } else if (colleague == colleagueB) {
      colleagueA.receive(message);
    }
  }
}

abstract class Colleague {
  protected Mediator mediator;

  Colleague(Mediator mediator) {
    this.mediator = mediator;
```



```
  }
}

class ColleagueA extends Colleague {
  ColleagueA(Mediator mediator) {
    super(mediator);
  }

  void send(String message) {
    mediator.notify(message, this);
  }

  void receive(String message) {
    System.out.println("Colleague A received: " + message);
  }
}

class ColleagueB extends Colleague {
  ColleagueB(Mediator mediator) {
    super(mediator);
  }

  void send(String message) {
    mediator.notify(message, this);
  }

  void receive(String message) {
    System.out.println("Colleague B received: " + message);
  }
}

public class MediatorExample {
  public static void main(String[] args) {
    ConcreteMediator mediator = new ConcreteMediator();

    ColleagueA a = new ColleagueA(mediator);
    ColleagueB b = new ColleagueB(mediator);

    mediator.setColleagues(a, b);

    a.send("Hello from A");
    b.send("Hello from B");
  }
}
```

Output:

```
Colleague B received: Hello from A
Colleague A received: Hello from B
```

9.1.8 Behavioral Design Patterns

9.1.8.0.1 Strategy Pattern Encapsulates algorithms and makes them interchangeable.


```
interface Strategy {
  void execute();
}

class ConcreteStrategyA implements Strategy {
  public void execute() {
    System.out.println("Executing Strategy A");
  }
}

class Context {
  private Strategy strategy;

  Context(Strategy strategy) {
    this.strategy = strategy;
  }

  void setStrategy(Strategy strategy) {
    this.strategy = strategy;
  }

  void executeStrategy() {
    strategy.execute();
  }
}

public class StrategyExample {
  public static void main(String[] args) {
    Context context = new Context(new ConcreteStrategyA());
    context.executeStrategy();
  }
}
```

9.1.8.0.2 Observer Pattern

Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.

```
import java.util.ArrayList;
import java.util.List;

interface Observer {
  void update(String message);
}

class Subject {
  private List<Observer> observers = new ArrayList<>();

  void addObserver(Observer observer) {
    observers.add(observer);
  }

  void notifyObservers(String message) {
    for (Observer observer : observers) {
      observer.update(message);
    }
  }
}
```



```
class ConcreteObserver implements Observer {
  public void update(String message) {
    System.out.println("Received: " + message);
  }
}

public class ObserverExample {
  public static void main(String[] args) {
    Subject subject = new Subject();
    Observer observer = new ConcreteObserver();

    subject.addObserver(observer);
    subject.notifyObservers("Event occurred");
  }
}
```

9.1.8.0.3 Chain of Responsibility Passes a request along a chain of handlers until one handles it.

```
abstract class Handler {
  private Handler next;

  public void setNext(Handler next) {
    this.next = next;
  }

  public void handleRequest(String request) {
    if (next != null) {
      next.handleRequest(request);
    }
  }
}

class ConcreteHandlerA extends Handler {
  @Override
  public void handleRequest(String request) {
    if (request.equals("A")) {
      System.out.println("Handler A handled the request");
    } else {
      super.handleRequest(request);
    }
  }
}

class ConcreteHandlerB extends Handler {
  @Override
  public void handleRequest(String request) {
    if (request.equals("B")) {
      System.out.println("Handler B handled the request");
    } else {
      super.handleRequest(request);
    }
  }
}
```



```
public class ChainOfResponsibilityExample {
  public static void main(String[] args) {
    Handler handlerA = new ConcreteHandlerA();
    Handler handlerB = new ConcreteHandlerB();

    handlerA.setNext(handlerB);

    handlerA.handleRequest("A");
    handlerA.handleRequest("B");
    handlerA.handleRequest("C");
  }
}
```

Output:

```
Handler A handled the request
Handler B handled the request
```

9.1.8.0.4 Iterator Pattern Provides a way to access elements of a collection sequentially without exposing the underlying representation.

```
import java.util.*;

class CustomCollection<T> {
  private List<T> items = new ArrayList<>();

  public void add(T item) {
    items.add(item);
  }

  public Iterator<T> iterator() {
    return items.iterator();
  }
}

public class IteratorExample {
  public static void main(String[] args) {
    CustomCollection<String> collection = new CustomCollection<>();
    collection.add("Item1");
    collection.add("Item2");
    collection.add("Item3");

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

Output:

```
Item1
Item2
Item3
```


9.1.8.0.5 Composite Pattern Composes objects into tree structures to represent part-whole hierarchies.

```
import java.util.*;

interface Component {
  void display();
}

class Leaf implements Component {
  private String name;

  public Leaf(String name) {
    this.name = name;
  }

  public void display() {
    System.out.println(name);
  }
}

class Composite implements Component {
  private List<Component> children = new ArrayList<>();

  public void add(Component component) {
    children.add(component);
  }

  public void display() {
    for (Component child : children) {
      child.display();
    }
  }
}

public class CompositeExample {
  public static void main(String[] args) {
    Composite root = new Composite();
    root.add(new Leaf("Leaf1"));
    Composite subtree = new Composite();
    subtree.add(new Leaf("Leaf2"));
    subtree.add(new Leaf("Leaf3"));
    root.add(subtree);

    root.display();
  }
}
```

Output:

```
Leaf1
Leaf2
Leaf3
```


9.1.9 Summary of Design Patterns and Best Practices

9.1.9.1 Creational Patterns

These patterns focus on object creation mechanisms:

- **Singleton Pattern**
  - Ensures a class has only one instance
  - Provides global point of access
  - Common in configuration managers and thread pools
- **Factory Pattern**
  - Creates objects without exposing creation logic
  - Uses common interface for object creation
  - Ideal for creating families of related objects
- **Builder Pattern**
  - Constructs complex objects step by step
  - Separates construction from representation
  - Excellent for objects with many optional parameters

9.1.9.2 Structural Patterns

These patterns establish relationships between objects:

- **Composite Pattern**
  - Treats individual objects and compositions uniformly
  - Creates tree-like hierarchies
  - Common in UI component hierarchies
- **Facade Pattern**
  - Provides unified interface to complex subsystem
  - Reduces coupling between subsystems
  - Simplifies client interaction with complex systems

- **Adapter Pattern**
  - Allows incompatible interfaces to work together
  - Converts interface into another interface client expects
  - Useful for integrating legacy systems
- **Decorator Pattern**
  - Adds behavior to objects dynamically
  - Alternative to subclassing for extending functionality
  - Common in I/O stream implementations

9.1.9.3 Behavioral Patterns

These patterns manage algorithms, relationships, and responsibilities between objects:

- **Chain of Responsibility**
  - Passes requests along a chain of handlers
  - Each handler decides to process or pass along
  - Common in event handling systems
- **Iterator Pattern**
  - Provides way to access elements sequentially
  - Hides underlying representation
  - Standard in collection frameworks
- **Observer Pattern**
  - Defines one-to-many dependency between objects
  - Automatically notifies dependents of state changes
  - Common in event handling systems
- **Strategy Pattern**
  - Defines family of algorithms
  - Makes algorithms interchangeable
  - Useful for varying behavior at runtime

9.1.9.4 Implementation Guidelines

- **Pattern Selection**
  - Choose patterns based on specific problem requirements
  - Consider maintainability and complexity trade-offs
  - Avoid over-engineering with unnecessary patterns
- **Implementation Considerations**
  - Keep implementations as simple as possible
  - Document pattern usage and rationale
  - Consider impact on testing and debugging
- **Pattern Combination**
  - Combine patterns when appropriate
  - Ensure patterns work together harmoniously
  - Maintain clean separation of concerns

9.1.9.5 Common Anti-Patterns to Avoid

- Overuse of Singleton pattern
- Complex inheritance hierarchies
- God objects that do too much
- Tight coupling between components
- Pattern-driven instead of requirement-driven design

Chapter 10

Java Language: Advanced Concepts

10.1 Streams and Lambda Expressions

Java introduced Streams and Lambda Expressions in Java 8 as part of the functional programming paradigm. Streams provide a clean and efficient way to process collections of data, while lambda expressions simplify the implementation of functional interfaces.

This chapter explores Stream pipelines (`filter`, `map`, `reduce`), the use of functional interfaces, and parallel streams for performance improvements.

10.1.1 Stream Pipelines: Filter, Map, Reduce

10.1.1.0.1 What are Streams? Streams are sequences of elements that support operations to transform, filter, and aggregate data. Streams do not modify the original data but produce a new result.

10.1.1.0.2 Stream Pipeline A Stream pipeline consists of three main components:

- Source: Collection, array, or I/O channel.
- Intermediate Operations: Transform the stream (e.g., `filter()`, `map()`).
- Terminal Operations: Produce a result (e.g., `collect()`, `reduce()`).

10.1.1.0.3 Example: Using `filter()` and `map()` The `filter()` method selects elements that match a condition, while the `map()` method transforms each element.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamExample {
  public static void main(String[] args) {
    List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "Anna");

    // Filter names starting with "A" and convert to uppercase
    List<String> result = names.stream()
    .filter(name -> name.startsWith("A")) // Intermediate operation
    .map(String::toUpperCase)            // Intermediate operation
    .collect(Collectors.toList());       // Terminal operation

    System.out.println("Filtered and Mapped List: " + result);
  }
}
```

Output:

```
Filtered and Mapped List: [ALICE, ANNA]
```


10.1.1.0.4 Example: Using `reduce()` for Aggregation

The `reduce()` method combines stream elements to produce a single result.

```
import java.util.Arrays;
import java.util.List;

public class ReduceExample {
  public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

    // Calculate the sum of all elements
    int sum = numbers.stream()
    .reduce(0, (a, b) -> a + b); // Accumulator function

    System.out.println("Sum: " + sum);
  }
}
```

Output:

```
Sum: 15
```

10.1.2 Using Functional Interfaces

10.1.2.0.1 What are Functional Interfaces?

A functional interface is an interface with exactly one abstract method. It is used as a target type for lambda expressions and method references.

10.1.2.0.2 Built-in Functional Interfaces in Java

The Java Standard Library provides several functional interfaces:

- **`Predicate<T>`**: Accepts an input and returns a boolean (`test()`).
- **`Function<T, R>`**: Accepts an input and returns a result (`apply()`).
- **`Consumer<T>`**: Accepts an input and performs an action (`accept()`).
- **`Supplier<T>`**: Provides a result without input (`get()`).

10.1.2.0.3 Example: Using Built-in Functional Interfaces

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class FunctionalInterfaceExample {
  public static void main(String[] args) {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```



```
    // Predicate to filter even numbers
    Predicate<Integer> isEven = n -> n % 2 == 0;

    numbers.stream()
    .filter(isEven)
    .forEach(n -> System.out.println("Even: " + n));
  }
}
```

Output:

```
Even: 2
Even: 4
```

10.1.2.0.4 Custom Functional Interfaces You can define your own functional interface using the `@FunctionalInterface` annotation.

```
@FunctionalInterface
interface MathOperation {
  int operate(int a, int b);
}

public class CustomFunctionalInterface {
  public static void main(String[] args) {
    // Lambda expression implementing the functional interface
    MathOperation addition = (a, b) -> a + b;

    System.out.println("Addition: " + addition.operate(5, 3));
  }
}
```

Output:

```
Addition: 8
```

10.1.3 Parallel Streams for Performance

10.1.3.0.1 What are Parallel Streams? Parallel streams enable multi-threaded execution of stream operations, improving performance for large datasets. A parallel stream divides the workload into smaller tasks and processes them concurrently.

10.1.3.0.2 Example: Parallel Stream for Faster Processing

```
import java.util.Arrays;
import java.util.List;

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



```
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

    // Using parallel stream
    int sum = numbers.parallelStream()
    .filter(n -> n % 2 == 0)
    .mapToInt(n -> n * n) // Square the number
    .sum();

    System.out.println("Sum of squares of even numbers: " + sum);
  }
}
```

Output:

```
Sum of squares of even numbers: 220
```

10.1.3.0.3 Key Points for Parallel Streams:

- Parallel streams split data into multiple threads for faster computation.
- Use parallel streams when working with large datasets and CPU-intensive tasks.
- Be cautious of race conditions when modifying shared resources.

10.1.4 Best Practices for Streams and Lambda Expressions

10.1.4.0.1 Best Practices for Streams and Lambda Expressions

- Use streams for cleaner and more readable code when processing collections.
- Prefer `filter`, `map`, and `reduce` for declarative programming.
- Use parallel streams only when performance gains justify the additional complexity.
- Avoid side effects in stream operations (e.g., modifying external variables).
- Use functional interfaces like `Predicate`, `Function`, and `Consumer` to simplify lambda expressions.
- Streams provide a functional approach to processing collections, enabling operations like `filter()`, `map()`, and `reduce()`.
- Lambda expressions simplify the implementation of functional interfaces, improving code readability.

- Parallel streams can improve performance for large datasets but require careful usage.
- Functional interfaces (`Predicate`, `Function`, etc.) enable reusable and concise code.

10.2 Advanced Features of Java: Reflection and Annotations

Java provides advanced features like the Reflection API and Annotations to enhance flexibility, introspection, and declarative programming. These features enable dynamic inspection of classes, methods, and fields at runtime, as well as the ability to define custom metadata.

This chapter explores the Reflection API for class inspection and demonstrates how to create and use custom annotations.

10.2.1 Reflection API for Class Inspection

10.2.1.0.1 What is the Reflection API? The Reflection API in Java allows inspection and manipulation of classes, methods, fields, and constructors at runtime. It is part of the `java.lang.reflect` package.

10.2.1.0.2 Key Features of the Reflection API:

- Inspect a class at runtime (e.g., methods, fields, constructors).
- Create objects dynamically.
- Invoke methods and modify fields dynamically.

10.2.1.0.3 Example: Inspecting Class Information The following example demonstrates how to inspect a class's metadata using the Reflection API.

```
import java.lang.reflect.*;

class Person {
  private String name;
  private int age;

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

  public void displayInfo() {
    System.out.println("Name: " + name + ", Age: " + age);
  }
}

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



```
    // Load the Person class dynamically
    Class<?> clazz = Class.forName("Person");

    // Display class name
    System.out.println("Class Name: " + clazz.getName());

    // Display declared fields
    System.out.println("\nFields:");
    for (Field field : clazz.getDeclaredFields()) {
      System.out.println(" - " + field.getName() + " (Type: " + field.getType() + ")");
    }

    // Display declared methods
    System.out.println("\nMethods:");
    for (Method method : clazz.getDeclaredMethods()) {
      System.out.println(" - " + method.getName() + "()");
    }

    // Create an instance of Person using Reflection
    Constructor<?> constructor = clazz.getConstructor(String.class, int.class);
    Object personInstance = constructor.newInstance("Alice", 25);

    // Access and invoke a method dynamically
    Method displayMethod = clazz.getMethod("displayInfo");
    displayMethod.invoke(personInstance);
  }
}
```

Output:

```
Class Name: Person

Fields:
- name (Type: class java.lang.String)
- age (Type: int)

Methods:
- displayInfo()

Name: Alice, Age: 25
```

Explanation:

- `Class.forName()` loads the class dynamically.
- `getDeclaredFields()` retrieves all fields (public, private, etc.).
- `getDeclaredMethods()` retrieves all methods.
- `Constructor.newInstance()` creates an object dynamically.
- `Method.invoke()` invokes methods dynamically.

10.2.2 Creating Custom Annotations

10.2.2.0.1 What are Annotations? Annotations provide metadata about classes, methods, fields, or parameters. They can be processed at compile-time or runtime to influence program behavior. Annotations are defined using the `@interface` keyword.

10.2.2.0.2 Defining Custom Annotations

```
import java.lang.annotation.*;

// Define a custom annotation
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@interface MyAnnotation {
  String author();
  String date();
  int version() default 1;
}

// Annotated class and method
@MyAnnotation(author = "John Doe", date = "2024-01-01", version = 2)
class MyClass {

  @MyAnnotation(author = "Jane Smith", date = "2024-02-01")
  public void myMethod() {
    System.out.println("Executing myMethod...");
  }
}
```

10.2.2.0.3 Processing Custom Annotations Using Reflection The following example retrieves and processes the custom annotations.

```
import java.lang.reflect.*;

public class AnnotationProcessor {
  public static void main(String[] args) {
    try {
      // Load the class dynamically
      Class<?> clazz = MyClass.class;

      // Process class-level annotation
      if (clazz.isAnnotationPresent(MyAnnotation.class)) {
        MyAnnotation annotation = clazz.getAnnotation(MyAnnotation.class);
        System.out.println("Class-level Annotation:");
        System.out.println("Author: " + annotation.author());
        System.out.println("Date: " + annotation.date());
        System.out.println("Version: " + annotation.version());
      }

      // Process method-level annotation
      Method method = clazz.getMethod("myMethod");
      if (method.isAnnotationPresent(MyAnnotation.class)) {
```



```
            MyAnnotation annotation = method.getAnnotation(MyAnnotation.class);
            System.out.println("\nMethod-level Annotation:");
            System.out.println("Author: " + annotation.author());
            System.out.println("Date: " + annotation.date());
            System.out.println("Version: " + annotation.version());
        }

        // Invoke the method
        Object obj = clazz.getDeclaredConstructor().newInstance();
        method.invoke(obj);

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

Output:

```
Class-level Annotation:
Author: John Doe
Date: 2024-01-01
Version: 2

Method-level Annotation:
Author: Jane Smith
Date: 2024-02-01
Version: 1
Executing myMethod...
```

Explanation:

- `@Retention(RetentionPolicy.RUNTIME)`: The annotation is available at runtime.
- `@Target`: Specifies where the annotation can be applied (e.g., class, method).
- Custom metadata is retrieved using Reflection's `getAnnotation()`.
- The method annotated with the custom annotation is invoked dynamically.

10.2.3 Best Practices for Reflection and Annotations

- Use Reflection sparingly as it can impact performance and bypass compile-time safety.
- Always validate input when using Reflection to avoid security vulnerabilities.

- Avoid making private fields or methods accessible unless absolutely necessary.
- Use annotations for configuration, metadata, and declarative programming.
- Keep annotations lightweight and avoid including complex logic.
- Combine annotations with frameworks like Spring or JUnit for powerful processing.
- Use retention policies carefully (e.g., `RUNTIME` for runtime processing).
- Reflection API enables dynamic inspection and manipulation of classes, methods, and fields at runtime.
- Use Reflection to retrieve class metadata, create objects, and invoke methods dynamically.
- Annotations provide metadata about code elements and can be processed at runtime or compile-time.
- Custom annotations are created using `@interface` and processed using Reflection.
- Follow best practices to ensure Reflection and annotations are used efficiently and securely.

By mastering Reflection and Annotations, developers can build flexible, configurable, and metadata-driven Java applications.

