MASTERING JAVA

FROM FUNDAMENTALS TO ADVANCED TECHNIQUES

CODING HUBS

2023

Index

Introduction

Introduction to the Book purpose and target audience

This book serves as a comprehensive guide to mastering the Java programming language.

Its primary purpose is to provide readers with a solid foundation in Java and equip them with the necessary skills to develop robust and efficient applications.

The target audience includes aspiring programmers, students, and professionals who want to enhance their knowledge of Java or transition to Java development.

Importance and widespread use of Java in various domains

Java, being one of the most widely used programming languages globally, has garnered immense popularity and continues to flourish in the ever-evolving tech landscape. This versatile language finds applications in diverse domains, making it a go-to choice for developers across various industries.

Web Development:

Java has a strong foothold in web development. It provides developers with robust tools and frameworks for building dynamic and scalable web applications. Java Servlets and JavaServer Pages (JSP) allow for server-side processing and the generation of dynamic web content. Additionally, Java frameworks like Spring MVC and Struts provide powerful solutions for developing enterprise-grade web applications. Java's versatility in web development extends to content management systems (CMS) as well, where Java-based CMS platforms enable the creation and management of content-rich websites.

Mobile App Development:

Java's significance in the mobile app development space cannot be overstated. It powers the Android operating system, making it the language of choice for building Android applications. The Android Software Development Kit (SDK) provides developers with a comprehensive set of tools and APIs, enabling them to create feature-rich and user-friendly mobile applications. Java libraries and frameworks designed for Android development, such as Retrofit, Gson, and Dagger, further enhance the development process, allowing developers to build powerful and efficient mobile apps.

Enterprise Systems:

Java has established itself as a dominant force in the realm of enterprise systems. Its reliability, scalability, and extensive ecosystem of frameworks and libraries make it an ideal choice for developing mission-critical systems. Java Enterprise Edition (Java EE) provides a platform for building enterprise applications, offering features like transaction management, security, and scalability. Enterprise frameworks like Spring and Hibernate, built on top of Java, simplify the development process and facilitate the creation of robust and maintainable enterprise applications.

Scientific Research:

Java's usage extends beyond traditional web and enterprise applications. It has found its way into the realm of scientific research and computational domains. Java's rich collection of libraries, such as Apache Commons Math and Weka, enable scientists and researchers to perform complex computations, data analysis, and machine learning tasks. Furthermore, Java's cross-platform compatibility and support for parallel programming make it a valuable tool for high-performance computing and scientific simulations.

Other Domains:

Java's wide-ranging applications are not limited to the aforementioned domains. It also holds a prominent place in various other areas. In the realm of game development, Java frameworks like LibGDX and jMonkeyEngine provide a solid foundation for creating games across different platforms. Java's platform independence and security features make it a viable choice for developing applications in the finance and banking sector. JavaFX, a modern UI framework, enables developers to build desktop applications with rich graphical user interfaces. Additionally, Java is widely used in educational software and e-learning platforms, facilitating the creation of interactive and engaging learning experiences.

Comparison of Java with other programming languages

Features	Java	Python	C++	JavaScript	C#
Type System	Static	Dynamic	Static	Dynamic	Static
Object-Oriented	Yes	Yes	Yes	Yes	Yes
Platform Independence	Yes	No	No	No	No
Memory Management	Garbage Collection	Garbage Collection	Manual	Garbage Collection	Garbage Collection
Concurrency Support	Yes	Yes	Yes	Yes	Yes
Web Development	Yes	Yes	Yes	Yes	Yes

Mobile Development	Yes	Yes	Yes	Yes	Yes (for Xamarin)
Community and Libraries	Large	Large	Large	Large	Large
Learning Curve	Moderate	Easy	Moderate	Easy	Moderate
Performance	Good	Moderate	Excellent	Good	Good
Popular Use Cases	Enterprise systems, Android development	Web development, Data analysis, Automation	System-level programming, Game development	Web development, Front-end development	Windows application development

Overview of the Java programming language and its key features

Java is an object-oriented programming language that emphasizes code reusability, modularity, and platform independence.

It provides a robust and secure programming environment with features like automatic memory management (garbage collection) and strong type checking.

Java's key features include

Java, a widely used programming language, possesses a rich set of key features that make it a preferred choice for developers across various domains. From its simplicity and object-oriented nature to its platform independence and robustness, Java offers a comprehensive toolkit for building high-performance, secure, and scalable applications. Let's delve into Java's key features in more detail.

Simple:

Java is designed to be a simple and straightforward language, making it accessible to beginners and experienced developers alike. Its syntax is clean and easy to read, reducing the learning curve and enabling developers to write code efficiently. Java's simplicity promotes code maintainability and enhances the productivity of developers.

Object-Oriented:

Java follows the object-oriented programming (OOP) paradigm, enabling developers to build modular, reusable, and scalable code. Objects are the fundamental building blocks in Java, encapsulating data and behavior within classes. Inheritance, polymorphism, and encapsulation are key concepts that allow for code organization, abstraction, and code reusability, fostering efficient software development.

Portable:

Java's portability is a significant advantage, allowing applications to run on different platforms without the need for modification. This is achieved through Java's "write once, run anywhere" principle. Java source code is compiled into bytecode, which can be executed on any platform with a Java Virtual Machine (JVM). This portability reduces development time and effort by eliminating the need to rewrite code for different operating systems.

Platform Independent:

Java's platform independence is a result of its bytecode execution on the JVM. By decoupling the code from the underlying hardware and operating system, Java applications can run consistently across various platforms. This feature enables seamless deployment and distribution of Java applications, making it an ideal choice for cross-platform development.

Secured:

Security is paramount in modern software development, and Java provides robust mechanisms to build secure applications. Java's security features include a comprehensive security architecture, sandboxing to restrict untrusted code, and a security manager for fine-grained access control. Java's built-in security ensures that applications can handle sensitive data, protect against vulnerabilities, and mitigate security risks effectively.

Robust:

Java's robustness stems from its strong type checking, automatic memory management through garbage collection, and exceptional error handling mechanisms. The Java Virtual Machine (JVM) detects and handles runtime errors, preventing application crashes. Java's robustness enhances application stability and reliability, providing a solid foundation for building enterprise-grade software.

Architecture Neutral:

Java's architecture-neutral nature allows it to run on various hardware and operating systems seamlessly. Java compilers generate bytecode, which is independent of the underlying architecture. This bytecode is then interpreted by the JVM, enabling Java applications to execute efficiently on different platforms without requiring recompilation. This feature simplifies application deployment and enables broad compatibility.

Interpreted:

Java is an interpreted language, meaning that Java bytecode is executed by the JVM at runtime. This interpretation process provides flexibility and enables dynamic runtime behavior. It allows for late binding, runtime code modification, and dynamic class loading, facilitating adaptability and extensibility in Java applications.

High Performance:

Despite being an interpreted language, Java offers high-performance capabilities. The JVM's Just-In-Time (JIT) compilation optimizes bytecode into machine code, improving execution speed. Java's performance is further enhanced by its efficient memory management system, multithreading capabilities, and optimizations provided by the JVM, making it suitable for demanding applications that require high throughput and low latency.

Multithreaded:

Java natively supports multithreading, allowing developers to write concurrent programs that can execute multiple threads simultaneously. Multithreading enhances application responsiveness, enables efficient utilization of modern multicore processors, and promotes parallel processing. Java's multithreading

capabilities facilitate the development of highly scalable and responsive applications.

Distributed:

Java provides extensive support for distributed computing through its robust networking libraries and APIs. Developers can build distributed applications, such as client-server systems or distributed computing frameworks, using Java's socket programming and Remote Method Invocation (RMI). Java's distributed computing features enable seamless communication and collaboration between distributed components.

Dynamic:

Java embraces dynamic programming through features like reflection and dynamic class loading. Reflection allows developers to inspect and modify classes and objects at runtime, enabling advanced runtime behaviors and dynamic code generation. Dynamic class loading allows developers to load classes dynamically, extending the flexibility and adaptability of Java applications.

Chapter 1: Introduction to Java Programming

1.1 Brief history and evolution of Java

Java, developed by James Gosling and his team at Sun Microsystems (later acquired by Oracle Corporation), has undergone a remarkable evolution since its initial release in 1995. From its humble beginnings as a language for programming consumer electronics to its current status as one of the most widely used and influential programming languages, Java has continued to evolve and adapt to the changing needs of the software development industry.

Here is a timeline of the key milestones in the evolution of Java:

Java 1.0 (1996): The first official release of Java introduced a groundbreaking concept - "Write Once, Run Anywhere." Java's platform independence, achieved through the use of the Java Virtual Machine (JVM), allowed developers to write code once and run it on any platform that had a compatible JVM. This portability made Java an attractive choice for cross-platform development.

Java 2 (1998): Java 2, also known as Java Development Kit (JDK) 1.2, brought significant improvements to the language, including the introduction of the Swing graphical user interface (GUI) framework, which replaced the original Abstract Window Toolkit (AWT). Java 2 also introduced new APIs, enhanced performance, and better security features.

Java SE (Standard Edition) (2004): With the release of Java SE, the Java platform was divided into three editions: Java SE, Java EE (Enterprise Edition), and Java

ME (Micro Edition). Java SE became the foundation for general-purpose Java applications, providing core libraries, tools, and the Java Runtime Environment (JRE) for running Java programs.

Java EE (Enterprise Edition) (1999): Java EE was developed to address the specific needs of enterprise-level software development. It provided a set of standardized APIs and frameworks for building robust and scalable enterprise applications. Java EE included technologies such as Java Servlets, JavaServer Pages (JSP), Enterprise JavaBeans (EJB), and Java Persistence API (JPA).

Java ME (Micro Edition) (1999): Java ME was designed for developing applications for resource-constrained devices, such as mobile phones, embedded systems, and set-top boxes. It provided a scaled-down version of the Java platform, tailored to the specific requirements of these devices.

Java 5 (2004): Java 5, also known as Java Development Kit (JDK) 1.5, introduced several significant features, including the addition of generics, enhanced support for annotations, improved concurrency with the java.util.concurrent package, and the introduction of the Java Virtual Machine (JVM) support for scripting languages through the Java Native Interface (JNI).

Java 8 (2014): Java 8 brought substantial changes to the language with the introduction of lambda expressions, functional interfaces, and the Stream API. These additions greatly enhanced Java's capabilities for functional programming and simplifying code that involves collections and data processing. Java 8 also introduced the new date and time API (java.time) and the Nashorn JavaScript engine.

Java 9 (2017): Java 9 introduced the Java Platform Module System (JPMS), also known as Project Jigsaw. JPMS brought modularity to the Java platform, allowing

developers to create more scalable and maintainable applications. It also introduced the Flow API for reactive programming and the new HttpClient API for more efficient HTTP communication.

Java 10 and Beyond: Subsequent releases of Java have focused on smaller, incremental updates rather than major language changes. These updates have included features such as local variable type inference (var), enhancements to the garbage collection system, performance improvements, and new APIs for working with reactive streams.

Java's evolution has not been limited to the language itself. The Java ecosystem has grown immensely, with a vast array of frameworks, libraries, and tools built around the language. Frameworks such as Spring, Hibernate, and Apache Struts have become integral parts of enterprise application development using Java.

Additionally, the Java community has played a pivotal role in shaping the language's evolution. The community-driven Java Community Process (JCP) allows developers, organizations, and experts to propose and contribute to changes in the Java language and platform specifications.

1.2 Introduction to the Java Virtual Machine (JVM) and its role in platform independence

The Java Virtual Machine (JVM) is a fundamental component of the Java platform. It plays a crucial role in enabling Java's platform independence, which is one of the language's defining features. Understanding the JVM and its role is essential to comprehend why Java code can run on different operating systems and hardware architectures without requiring recompilation.

What is the JVM?

The JVM is a software implementation of a virtual machine that executes Java bytecode. It acts as an intermediary between Java code and the underlying operating system and hardware. When a Java program is compiled, the resulting bytecode is platform-independent and can be executed on any system that has a compatible JVM installed.

The JVM performs several critical tasks:

Bytecode Execution: The JVM reads and executes Java bytecode instructions, which are compact and low-level representations of Java code. It interprets the bytecode instructions or, in some cases, dynamically compiled them into native machine code for improved performance.

Memory Management: The JVM manages memory allocation and deallocation for Java programs. It automatically handles memory allocation for objects and performs garbage collection to reclaim memory occupied by objects that are no longer in use. This automatic memory management alleviates the burden of manual memory allocation and deallocation, reducing the risk of memory leaks and dangling pointers.

Security: The JVM includes security features that protect the Java runtime environment and applications. It provides a sandboxing mechanism that isolates untrusted code from the underlying system, preventing it from accessing sensitive resources or causing harm. The JVM's security manager allows fine-grained control over permissions, ensuring that applications adhere to specified security policies.

Platform Adaptation: The JVM abstracts the underlying hardware and operating system, allowing Java programs to run consistently across different platforms. When the JVM is installed on a specific platform, it provides an implementation of

the Java runtime environment tailored for that platform. This abstraction shields Java applications from platform-specific details, enabling them to run without modifications.

Platform Independence and the JVM:

The key factor that enables platform independence in Java is the JVM's ability to execute bytecode across various platforms. When Java source code is compiled, it is transformed into bytecode, which is a machine-independent representation of the code. The bytecode is then interpreted or compiled by the JVM, which is specific to the target platform. This approach decouples the Java code from the intricacies of the underlying hardware and operating system.

The JVM's platform independence provides several advantages:

Portability: Java applications written once can be executed on any platform that has a compatible JVM installed. This eliminates the need for rewriting or recompiling code for different platforms, saving time and effort in software development and deployment.

Write Once, Run Anywhere (WORA): The JVM's platform independence aligns with the "Write Once, Run Anywhere" principle of Java. It allows developers to write code once and deploy it on multiple platforms, reducing the cost and complexity of cross-platform development.

Consistent Behavior: Java applications exhibit consistent behavior across platforms since they are executed within the JVM's controlled runtime environment. This uniformity ensures that Java programs behave predictably regardless of the underlying system, enhancing reliability and maintainability.

Ecosystem Compatibility: The JVM's platform independence extends beyond the Java language itself. Other languages like Kotlin, Scala, and Groovy can also be compiled into bytecode and executed on the JVM, benefiting from the same platform independence and ecosystem of libraries and tools.

It's worth noting that while the JVM enables platform independence, Java libraries and frameworks used in development may have platform-specific dependencies or optimizations. However, the core logic of the Java code remains platform-independent, and the JVM ensures its execution across different platforms.

1.3 Difference between JDK, JRE, and JVM

JDK (Java Development Kit):

The JDK (Java Development Kit) is a software development kit that provides the necessary tools and resources for developing Java applications. It contains everything required for the complete development cycle, including writing, compiling, and debugging Java code. The key components of JDK include:

Java Compiler (javac): The JDK includes the Java compiler, which translates Java source code into bytecode, a platform-independent representation of the code.

Java Runtime Environment (JRE): The JDK includes the JRE, allowing developers to run Java applications during development and testing. It provides the necessary runtime libraries and environment to execute Java programs.

Development Tools: The JDK provides a set of development tools, such as debuggers, profilers, and build automation tools (like Ant or Maven), to aid in the development process.

API Libraries: The JDK includes a comprehensive set of libraries and APIs (Application Programming Interfaces) that provide pre-built functionality for common tasks, such as I/O operations, networking, database connectivity, and GUI development.

JRE (Java Runtime Environment):

The JRE (Java Runtime Environment) is an essential part of the Java platform. It provides the runtime environment necessary to execute Java applications. The JRE includes the following components:

Java Virtual Machine (JVM): The JRE includes the JVM, which is responsible for executing Java bytecode. It provides an abstraction layer between the Java application and the underlying hardware and operating system.

Core Libraries: The JRE includes a set of core libraries that provide common functionality and services to Java applications, such as memory management, security, and thread management.

Java Class Libraries: The JRE includes a collection of class libraries, also known as the Java API (Application Programming Interface), which offers a vast range of pre-built classes and methods for various tasks, including file I/O, networking, GUI development, and database connectivity.

JVM (Java Virtual Machine):

The JVM (Java Virtual Machine) is the runtime environment for executing Java applications. It is responsible for interpreting or compiling Java bytecode into

machine-specific instructions that can be executed by the underlying hardware. The JVM provides the following functionalities:

Memory Management: The JVM manages memory allocation and deallocation through techniques like automatic garbage collection. It frees developers from manual memory management tasks, reducing the risk of memory leaks and memory-related bugs.

Platform Independence: The JVM ensures platform independence by executing Java bytecode, which is a platform-neutral representation of the Java code. It abstracts the underlying hardware and operating system, allowing Java applications to run consistently on different platforms.

Just-In-Time (JIT) Compilation: The JVM includes a Just-In-Time (JIT) compiler that dynamically translates frequently executed bytecode into native machine code for improved performance. This compilation technique optimizes the execution of Java applications.

1.4 Writing and running your first Java program

It's time to write your first Java program, the classic "Hello, World!" example.

Launch your preferred text editor or IDE and create a new Java source file with the extension ".java."

Using a text editor, create a file named "HelloWorld.java" and save it in a directory of your choice.

Write the Java code for the "Hello, World!" program, which consists of a class with a main method that prints the message to the console:

Example:

Open the "HelloWorld.java" file and write the following code:

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

Save the file once you have written the code.

To compile the Java program, open a command prompt or terminal, navigate to the directory where the "HelloWorld.java" file is located, and use the javac command followed by the filename

Chapter 2: Java Syntax and Data Types

2.1 Variables and their types in Java

Variables play a crucial role in programming as they allow us to store and manipulate data. In Java, there are three main types of variables: local variables, instance variables, and static variables. Let's delve into each type in detail:

Local Variables:
Local variables are declared within a method, constructor, or block and are accessible only within their declared scope. They are created when the method or block is entered and destroyed when the method or block exits. Local variables must be initialized before they are used.

Key points about local variables:

- They are declared within a method, constructor, or block using the appropriate data type.
- They exist on the stack memory.
- Local variables are not accessible from other methods or blocks in the class.
- Each time the method or block is called or executed, a new copy of the local variables is created.

Example:

```
public void exampleMethod() {
    int num = 10; // Local variable
    System.out.println(num);
}
```

Instance Variables:

Instance variables, also known as non-static variables, are declared within a class but outside any method, constructor, or block. They are associated with an instance of the class and each instance of the class has its own copy of instance variables. Instance variables are initialized with default values if not explicitly assigned.

Key points about instance variables:

- They are declared within a class but outside any method, constructor, or block.
- Instance variables exist as long as the instance of the class exists (i.e., as long as the object is not garbage collected).
- Instance variables are accessible by all methods, constructors, and blocks in the class.
- Each instance of the class has its own set of instance variables.

Example:

```
public class ExampleClass {
    int age; // Instance variable

    public void setAge(int newAge) {
        age = newAge;
    }

    public void displayAge() {
```

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

Static Variables (Class Variables):

Static variables, also known as class variables, are declared within a class but outside any method, constructor, or block, and are marked with the static keyword. They are associated with the class itself rather than with instances of the class. Static variables are shared among all instances of the class and are initialized with default values if not explicitly assigned.

Key points about static variables:

- They are declared within a class but outside any method, constructor, or block, and are marked with the static keyword.
- Static variables exist as long as the class is loaded into memory.
- Static variables are accessible by all instances of the class as well as by the class itself.
- Only a single copy of static variables exists, regardless of the number of instances created.

Example:

```
public class ExampleClass {
    static int totalCount; // Static variable

    public ExampleClass() {
        totalCount++;
    }

    public static void displayTotalCount() {
        System.out.println(totalCount);
    }
}
```

2.2 Declaring variables and initializing values

In Java, declaring variables involves specifying the variable's name, type, and optionally, an initial value. Initializing a variable refers to assigning it an initial value at the time of declaration. Here's a detailed explanation of declaring variables and initializing values

Variable Declaration:

Variable declaration involves specifying the variable's name and type. It informs the compiler about the existence of a variable without allocating memory for it. The general syntax for declaring a variable is:

```
type variableName;
```

Example:

```
int age;
double salary;
String name;
```

Variable Initialization:

Variable initialization refers to assigning an initial value to a declared variable. It can be done at the time of declaration or later in the program. Initialization ensures that a variable has a known value before it is used. The syntax for initializing a variable is:

```
type variableName = initialValue;
```

Example:

```
int age = 25;
double salary = 50000.5;
String name = "John";
```

Default Values:

In Java, variables are assigned default values if they are not explicitly initialized. The default values depend on the variable's type:
Numeric types (byte, short, int, long, float, double): 0 or 0.0
Boolean type (boolean): false
Character type (char): '\u0000' (null character)
Reference types (classes, arrays, interfaces): null

```
int count; // default value: 0
double price; // default value: 0.0
boolean isTrue; // default value: false
String message; // default value: null
```

Multiple Variable Declaration:

Java allows declaring multiple variables of the same type in a single statement. Each variable must be separated by a comma (,). This is known as variable declaration with initialization or variable list initialization.

Example:

```
int a = 1, b = 2, c = 3;
String firstName = "John", lastName = "Doe";
```

Final Variables:

The final keyword is used to declare constants in Java. Final variables cannot be reassigned once initialized. They must be assigned a value during declaration or within the constructor if declared as an instance variable. Final variables are typically written in uppercase letters.

Example:

```
final double PI = 3.14159;
final int MAX_VALUE;
MAX_VALUE = 100;
```

Variable Naming Rules:

In Java, variables must follow certain naming rules:

- Variable names are case-sensitive.
- The first character must be a letter, underscore (_), or dollar sign ($).
- Subsequent characters can be letters, digits, underscores, or dollar signs.
- Java keywords cannot be used as variable names.
- Variable names should be descriptive and meaningful.

Example:

```
int studentCount;
double averageSalary;
String firstName;
```

2.3 Operators in Java

Operator in Java is a symbol that is used to perform operations. For example: +, -, *, / etc. there are various types of operators

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator,
- Assignment Operator.

Unary Operator:

Unary operators act on a single operand and perform operations such as incrementing, decrementing, or negating its value. Examples include:

- Increment (++): Increases the value of the operand by 1.
- Decrement (--): Decreases the value of the operand by 1.
- Unary Plus (+): Indicates a positive value.
- Unary Minus (-): Negates the value of the operand.
- Logical NOT (!): Negates the boolean value of the operand.
- Bitwise Complement (~): Inverts the bits of the operand.

Arithmetic Operator:

Arithmetic operators perform mathematical operations on numeric operands. Examples include:

- Addition (+): Adds two operands.
- Subtraction (-): Subtracts the second operand from the first.
- Multiplication (*): Multiplies two operands.
- Division (/): Divides the first operand by the second.
- Modulus (%): Returns the remainder after division.

Shift Operator:

Shift operators shift the bits of the left operand by a specified number of positions. Examples include:

- Left Shift (<<): Shifts the bits to the left, filling with zeros.
- Right Shift (>>): Shifts the bits to the right, preserving the sign of the leftmost bit.
- Unsigned Right Shift (>>>): Shifts the bits to the right, filling with zeros.

Relational Operator:

Relational operators compare two operands and return a boolean value. Examples include:

- Equal to (==): Checks if the operands are equal.
- Not equal to (!=): Checks if the operands are not equal.
- Greater than (>), Less than (<), Greater than or equal to (>=), Less than or equal to (<=): Compare the values of the operands.

Bitwise Operator:

Bitwise operators perform operations on individual bits of integer values. Examples include:

- Bitwise AND (&): Performs a bitwise AND operation.

- Bitwise OR (|): Performs a bitwise OR operation.
- Bitwise XOR (^): Performs a bitwise XOR (exclusive OR) operation.

Logical Operator:

Logical operators perform logical operations on boolean expressions and return a boolean value. Examples include:

- Logical AND (&&): Returns true if both operands are true.
- Logical OR (||): Returns true if either of the operands is true.

Ternary Operator:

The ternary operator (? :) is a conditional operator that evaluates a boolean expression and returns one of two values based on the result.

Assignment Operator:

The assignment operator (=) assigns a value to a variable. Compound assignment operators (e.g., +=, -=) combine arithmetic operations with assignment.

Chapter 3: Control Structures and Decision Making

Control flow statements in Java allow programmers to control the flow of execution in a program based on specific conditions or to repeat certain code blocks. There are three types of control flow statements: decision-making statements, loop statements, and jump statements. Let's explore each type in detail, along with their syntax, a small program, and an explanation of the code:

1. Decision Making Statements

Decision-making statements allow the program to execute different blocks of code based on certain conditions.

1.1. if statements:

The if statement allows the program to execute a block of code if a given condition is true. It can be followed by an optional else statement to handle the case when the condition is false.

Syntax:

```
if (condition) {
    // Code to be executed if the condition is true
} else {
    // Code to be executed if the condition is false
}
```

Example:

```
int num = 10;
if (num > 0) {
    System.out.println("Number is positive");
} else {
    System.out.println("Number is non-positive");
}
```

Explanation:

In the above example, the if statement checks if the value of the variable num is greater than 0. If it is true, the program prints "Number is positive." Otherwise, it prints "Number is non-positive."

1.2. switch statement:

The switch statement evaluates an expression and executes different blocks of code based on different cases. It provides an alternative to long if-else chains.

Syntax:

```
switch (expression) {
    case value1:
        // Code to be executed if expression matches value1
        break;
    case value2:
        // Code to be executed if expression matches value2
        break;
    default:
        // Code to be executed if expression does not match any case
}
```

Example:

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

Explanation:

In the above example, the switch statement evaluates the value of the variable day. If it matches any of the cases, the corresponding block of code is executed. In this case, since day is 3, the program prints "Tuesday."

2. Loop Statements

Loop statements allow the program to repeat a block of code multiple times until a specific condition is met.

2.1. do-while loop:

The do-while loop executes a block of code at least once and then repeatedly executes the block as long as a given condition is true.

Syntax:

```
do {
    // Code to be executed
} while (condition);
```

Example:

```
int i = 1;
do {
    System.out.println(i);
    i++;
} while (i <= 5);
```

Explanation:

In the above example, the do-while loop prints the value of i and increments it by 1 in each iteration. It continues to execute the loop until the condition i <= 5 is no longer true.

2.2. while loop:

The while loop repeatedly executes a block of code as long as a given condition is true. It may not execute the block at all if the condition is initially false.

Syntax:

```
while (condition) {
    // Code to be executed
}
```

Example:

```
int i = 1;
while (i <= 5) {
    System.out.println(i);
    i++;
}
```

Explanation:

In the above example, the while loop prints the value of i and increments it by 1 in each iteration. It continues to execute the loop until the condition i <= 5 is no longer true.

2.3. for loop:

The for loop is used when the number of iterations is known in advance. It consists of an initialization, a condition, and an increment/decrement statement.

Syntax:

```
for (initialization; condition; update) {
    // Code to be executed
}
```

Example:

```
for (int i = 1; i <= 5; i++) {
    System.out.println(i);
}
```

Explanation:

In the above example, the for loop initializes i to 1, executes the loop as long as i is less than or equal to 5, and increments i by 1 in each iteration. It prints the value of i in each iteration.

2.4. for-each loop:

The for-each loop simplifies the process of iterating over elements in an array or a collection. It iterates over each element without the need for an index.

Syntax:

```
for (elementType element : arrayOrCollection) {
    // Code to be executed
}
```

Example:

```
int[] numbers = {1, 2, 3, 4, 5};
for (int number : numbers) {
    System.out.println(number);
}
```

Explanation:

In the above example, the for-each loop iterates over each element in the numbers array and assigns it to the variable number. It then prints the value of the number in each iteration.

3. Jump Statements:

Jump statements allow the program to control the flow of execution by jumping to specific points in the code.

3.1. Break Statement:

The break statement is used to exit from a loop or switch statement. It terminates the loop prematurely and continues execution after the loop.

Syntax:

```
break;
```

Example:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        break; // Exit the loop when i equals 3
    }
    System.out.println(i);
}
```

Explanation:

In the above example, the for loop iterates from 1 to 5. When the value of i becomes 3, the break statement is encountered, and the loop is terminated prematurely.

3.2. Continue Statement:

The continue statement is used to skip the remaining code in the current iteration of a loop and proceed to the next iteration.

Syntax:

```
continue;
```

Example:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue; // Skip the iteration when i equals 3
    }
    System.out.println(i);
}
```

Explanation:

In the above example, the for loop iterates from 1 to 5. When the value of i becomes 3, the continue statement is encountered, and the remaining code in that iteration is skipped. The loop then proceeds to the next iteration.

Chapter 4: Object-Oriented Programming in Java

4.1 Understanding the principles of object-oriented programming

4.2 OOPs Concept

OOPs (Object-Oriented Programming) is a programming paradigm that organizes code around objects, which have data (attributes) and behavior (methods). It emphasizes concepts like encapsulation, inheritance, and polymorphism to create modular and reusable code.

4.3 OOPs in java

OOPs in Java refers to the implementation of Object-Oriented Programming principles in the Java programming language. It involves organizing code around objects that encapsulate data and behavior. Key OOPs concepts in Java include encapsulation, inheritance, polymorphism, and abstraction. Java's built-in support for these concepts enables the creation of modular, reusable, and maintainable code.

4.4 List of OOPs Concepts in Java

- Objects
- Classes
- Abstraction
- Inheritance
- Polymorphism
- Encapsulation

4.4.1 Objects

In Object-Oriented Programming (OOP) in Java, an object is an instance of a class that represents a specific entity or concept. Objects have both state (attributes or data) and behavior (methods or operations) associated with them. They are the fundamental building blocks in Java that allow us to model and manipulate real-world entities within a program.

Let's understand Objects in Java with an example:

```
public class Student {
    private String name;
    private int age;

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

    public void displayDetails() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }

    public static void main(String[] args) {
        Student student1 = new Student("John Doe", 20);  // Creating
an object of the Student class
        student1.displayDetails();  // Calling the displayDetails()
method of the Student object
```

```
        Student student2 = new Student("Jane Smith", 22);  // Creating
another Student object
        student2.displayDetails();  // Calling the displayDetails()
method of the second Student object
    }}
```

In the example above, we have a Student class that represents a student object. It has attributes like name and age, and a method called displayDetails() that prints the student's information.

Within the main() method, we create two instances of the Student class, student1 and student2, using the new keyword and providing the necessary arguments to the constructor. Each object has its own set of attributes (name and age) that can hold different values.

We then call the displayDetails() method on each student object. This method prints out the name and age of the student to the console.

The use of objects allows us to create multiple instances of a class, each with its own unique set of attributes and behavior. We can interact with these objects, call their methods, and access their attributes to perform specific actions or retrieve information.

Objects provide a way to organize and structure code, enabling us to work with complex systems by breaking them down into smaller, manageable entities. They facilitate code reusability, modularity, and extensibility, as objects can be easily created and used in different parts of a program.

4.4.2 Classes

In Object-Oriented Programming (OOP) in Java, a class is a blueprint or template that defines the structure, behavior, and attributes of objects. It serves as a blueprint for creating multiple instances of objects with similar characteristics.

A class encapsulates data (attributes) and behavior (methods) that define the properties and actions of objects. It provides a way to define and organize related code and serves as a blueprint for creating objects.

Let's understand classes in Java with an example:

```
public class Rectangle {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    public double calculateArea() {
        return length * width;
    }

    public double calculatePerimeter() {
        return 2 * (length + width);
    }

    public static void main(String[] args) {
        Rectangle rectangle1 = new Rectangle(5.0, 3.0);  // Creating
an object of the Rectangle class
```

```
        double area = rectangle1.calculateArea();  // Calling the
calculateArea() method of the Rectangle object
        double perimeter = rectangle1.calculatePerimeter();  //
Calling the calculatePerimeter() method

        System.out.println("Area: " + area);
        System.out.println("Perimeter: " + perimeter);
    }
}
```

In the above example, we have a Rectangle class that represents a rectangle object. It has attributes like length and width, and methods like calculateArea() and calculatePerimeter().

Inside the main() method, we create an object rectangle1 of the Rectangle class using the new keyword and provide the necessary arguments to the constructor. This object has its own set of attributes that can hold specific values.

We then call the calculateArea() and calculatePerimeter() methods on the rectangle1 object. These methods perform calculations based on the length and width attributes of the rectangle and return the area and perimeter values.

The class serves as a blueprint for creating objects with similar attributes and behavior. It defines the structure and behavior of the objects, allowing us to create multiple instances of the class and interact with them independently.

Classes provide a way to organize and encapsulate related data and behavior, promoting code reusability and modularity. They enable us to create complex systems by defining the characteristics and actions of the objects within them.

4.4.3 Abstraction

Abstraction is a fundamental concept in Object-Oriented Programming (OOP) that allows us to model complex systems by focusing on the essential features while hiding unnecessary details. It is the process of representing the essential characteristics of an object or system without including the implementation details.

In Java, abstraction is achieved through abstract classes and interfaces. An abstract class is a class that cannot be instantiated and may contain abstract methods, which are declared without an implementation. Interfaces, on the other hand, define a contract for classes to implement certain methods without providing any implementation details themselves.

Let's understand abstraction in Java with an example:

```
abstract class Shape {
    public abstract void draw();
}

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

class Rectangle extends Shape {
    @Override
    public void draw() {
```

```
        System.out.println("Drawing a rectangle");
    }
}

public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle();
        circle.draw();  // Calling the draw() method of the Circle
object

        Shape rectangle = new Rectangle();
        rectangle.draw();  // Calling the draw() method of the
Rectangle object
    }
}
```

In the above example, we have an abstract class Shape that defines an abstract method draw(). This method is declared without any implementation, as it will be implemented in the concrete subclasses. The Circle and Rectangle classes extend the Shape class and provide their own implementation of the draw() method.

Inside the main() method, we create objects of the Circle and Rectangle classes, but we declare them as instances of the Shape class. This is possible because of abstraction, where we can treat objects of different classes as instances of a common superclass or interface.

By calling the draw() method on the circle and rectangle objects, we can invoke the respective implementations defined in the concrete subclasses. The exact implementation details of drawing a circle or a rectangle are hidden from the user, who only needs to know that they can call the draw() method on a Shape object.

Abstraction allows us to work at a higher level of abstraction, focusing on the essential characteristics and behaviors of objects without getting into the implementation details. It promotes code modularity, reusability, and maintainability, as changes to the implementation details of a class do not affect the code that uses the abstracted interface.

4.4.4 Inheritance

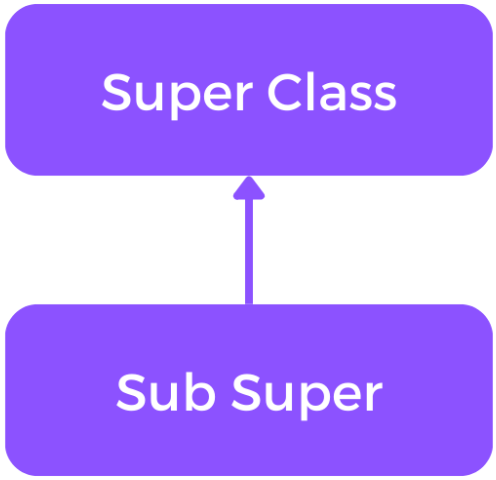
Inheritance is a key concept in Object-Oriented Programming (OOP) that allows a class to inherit properties and behaviors from another class. It enables code reuse, promotes modularity, and facilitates the creation of hierarchical relationships between classes.

Syntax:

```
class Subclass extends Superclass {
    // Subclass members
}
```

There are five types of inheritance:

4.4.4.1 Single Inheritance:



Single inheritance involves a subclass inheriting from a single superclass. It forms a direct parent-child relationship between two classes.

Syntax:

```
Class a {
…
}
Class b extends class a {
…
}
```

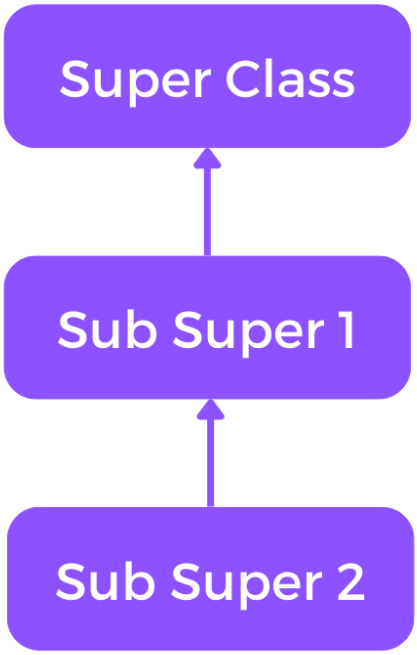
Example:

```
class Vehicle {
    protected String brand;
```

```
    public void setBrand(String brand) {
        this.brand = brand;
    }
}

class Car extends Vehicle {
    private int numOfDoors;

    public void setNumOfDoors(int numOfDoors) {
        this.numOfDoors = numOfDoors;
    }
}
```

In the example above, the Car class inherits from the Vehicle class. The Car class inherits the setBrand() method from the Vehicle class, and it adds its own unique member setNumOfDoors().

4.4.4.2 Multilevel Inheritance:



Multilevel inheritance involves a subclass becoming the superclass for another subclass. It forms a chain of inheritance with each class inheriting from its immediate superclass.

Syntax:

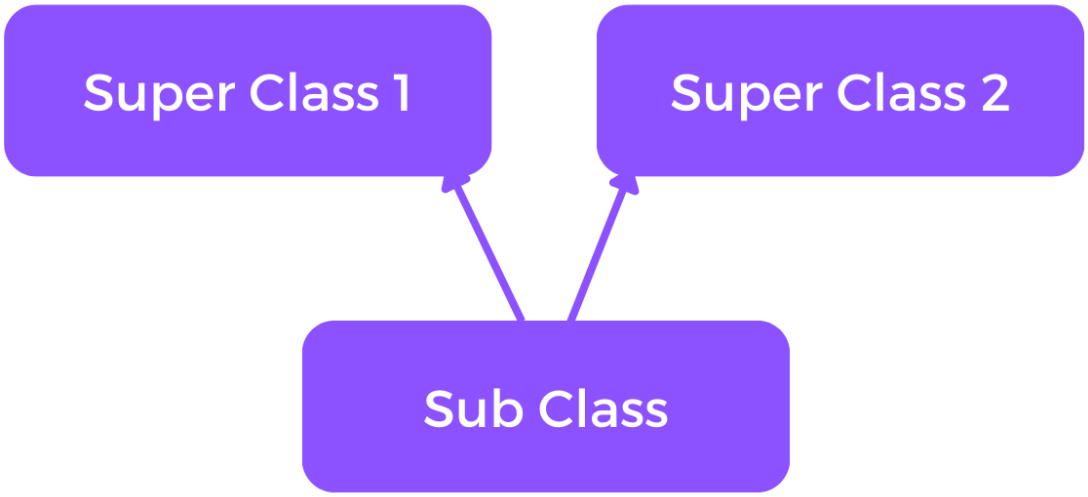
```
Class a {
….
}
Class b extends class a {
….
}
Class c extends class b {
…
}
```

Example:

```
class Animal {
    protected String name;

    public void setName(String name) {
        this.name = name;
    }
}

class Mammal extends Animal {
    public void displayInfo() {
        System.out.println("I am a mammal.");
    }
}

class Dog extends Mammal {
    public void bark() {
        System.out.println("Woof!");
    }
}
```

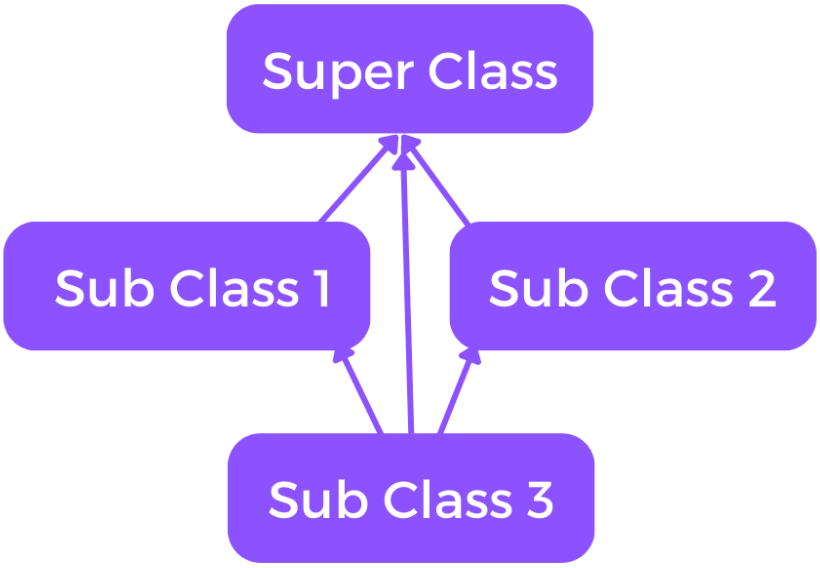
In this example, the Animal class is the superclass of the Mammal class, and the Mammal class is the superclass of the Dog class. Each class inherits the members of its immediate superclass.

4.4.4.3 Multiple Inheritance (Not supported in Java):



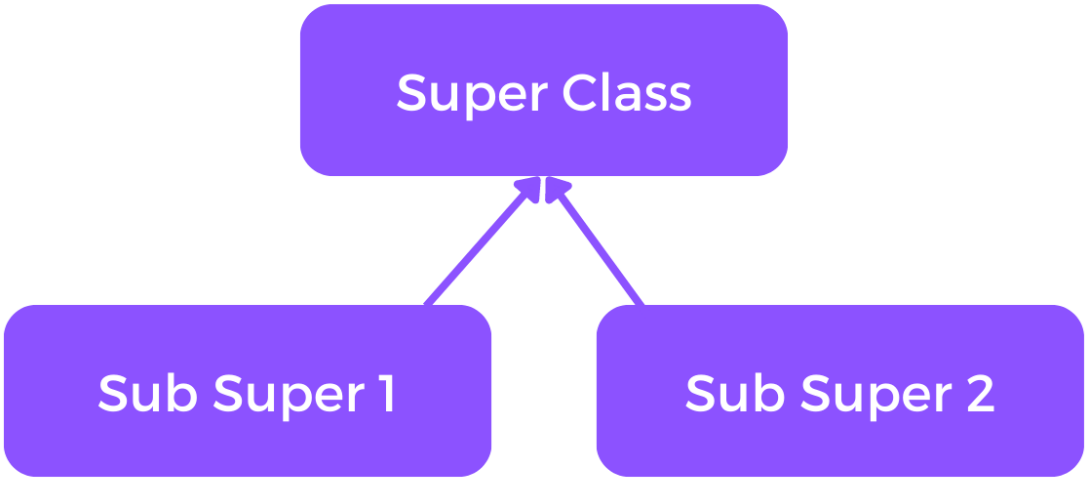
Multiple inheritance involves a subclass inheriting from multiple superclasses. However, Java does not support multiple inheritance of classes. It can be achieved using interfaces.

4.4.4.4 Hybrid Inheritance (Combination of multiple and hierarchical inheritance):



Hybrid inheritance involves a combination of inheritance types, such as multiple inheritance and hierarchical inheritance.

4.4.4.5 Hierarchical Inheritance:



Hierarchical inheritance occurs when multiple subclasses inherit from a single superclass. It forms a tree-like structure with a single superclass and multiple subclasses.

Syntax:

```
Class a {
…
}
Class b extends class a {
..
}
Class c extends class a {
..
}
```

Example:

```
class Animal {
    protected String name;

    public void setName(String name) {
        this.name = name;
    }
}

class Cat extends Animal {
    public void meow() {
        System.out.println("Meow!");
    }
}

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

In this example, both the Cat and Dog classes inherit from the Animal class. They share the setName() method from the Animal class and have their own unique methods.

4.4.5 Polymorphism

Polymorphism refers to many forms, or it is a process that performs a single action in different ways. It occurs when we have many classes related to each other by inheritance. Polymorphism is of two different types, i.e., compile-time polymorphism and runtime polymorphism. One of the examples of Compile time polymorphism is when we overload a static method in java. Run time

polymorphism also called a dynamic method dispatch is a method in which a call to an overridden method is resolved at run time rather than compile time. In this method, the overridden method is always called through the reference variable. By using method overloading and method overriding, we can perform polymorphism. Generally, the concept of polymorphism is often expressed as one interface, and multiple methods. This reduces complexity by allowing the same interface to be used as a general class of action.

Polymorphism in java can be classified into two types:

1. Static / Compile-Time Polymorphism
2. Dynamic / Runtime Polymorphism

4.4.5.1 Static / Compile-Time Polymorphism:

Static polymorphism occurs at compile-time and is also known as method overloading. It allows multiple methods with the same name but different parameters to coexist in a class.

Example:

```
class Calculator {
    public int add(int num1, int num2) {
        return num1 + num2;
    }

    public double add(double num1, double num2) {
        return num1 + num2;
    }
}
```

In the above example, the Calculator class has two add() methods with different parameter types. During compilation, the appropriate method is selected based on the argument types used when invoking the method.

4.4.5.2 Dynamic / Runtime Polymorphism

Dynamic polymorphism occurs at runtime and is also known as method overriding. It allows a subclass to provide a different implementation of a method defined in its superclass.

Example:

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

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

In this example, the Animal class has a makeSound() method, which is overridden in the Dog class. When calling the makeSound() method on a Dog object, the overridden implementation in the Dog class is invoked instead of the one in the Animal class.

4.4.6 Encapsulation

Encapsulation is a fundamental concept in Object-Oriented Programming (OOP) that involves bundling data (attributes) and methods (behavior) together within a class, and restricting access to the internal data of an object from outside the class. It encapsulates the data and provides controlled access to it through public methods, thereby protecting the integrity of the data and ensuring proper interaction with the object.

In Java, encapsulation is achieved through the use of access modifiers (public, private, protected) to control the visibility and accessibility of members (fields and methods) of a class.

Syntax:

```
class ClassName {
    private dataType fieldName;  // private field
    // Other fields and methods

    public returnType getFieldName() {
        // Getter method to access the private field
        // Perform any necessary operations
        return fieldName;
    }

    public void setFieldName(dataType value) {
        // Setter method to modify the private field
        // Perform any necessary operations
        fieldName = value;
    }
}
```

Here's an example to illustrate encapsulation in Java:

```
class BankAccount {
    private String accountNumber;  // Encapsulated private field
    private double balance;

    public String getAccountNumber() {
        return accountNumber;  // Getter method to access the private
field
    }

    public void setAccountNumber(String accountNumber) {
        // Setter method to modify the private field
        // Additional validations and business logic can be added here
        this.accountNumber = accountNumber;
    }

    public double getBalance() {
        return balance;  // Getter method to access the private field
    }

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
        } else {
            System.out.println("Insufficient balance.");
        }
```

```
    }
}
```

In this example, the BankAccount class encapsulates the accountNumber and balance fields, making them private to restrict direct access from outside the class. Instead, public getter and setter methods (getAccountNumber(), setAccountNumber(), getBalance()) are provided to access and modify the private fields.

By encapsulating the fields, we can control how they are accessed and manipulated. The getter methods provide read-only access to the accountNumber and balance fields, while the setter methods allow modification of the accountNumber. Additional validations and business logic can be added in the setter methods to ensure data integrity and maintain consistency.

Encapsulation ensures that the internal representation of an object is hidden from external access, promoting information hiding and abstraction. It protects the data from direct modification and provides a well-defined interface to interact with the object. This enhances code maintainability, reusability, and security.

4.4.7 Advantages of OOPs Concept

Object-Oriented Programming (OOP) brings several advantages that contribute to efficient and maintainable software development. Here are some key advantages of OOPs concepts:

4.4.7.1 Modularity and Code Reusability:

OOP promotes code modularity by encapsulating related data (attributes) and behavior (methods) within objects and classes. This modularity allows for easier code organization, maintenance, and reusability. Objects can be reused in

different parts of the program or in other projects, leading to faster development and reduced duplication of code.

4.4.7.2 Abstraction and Problem Solving:

OOP allows complex systems to be modeled and represented using simpler, abstract objects. Abstraction focuses on the essential features of an object while hiding unnecessary implementation details. It enhances problem-solving by breaking down a system into manageable and understandable components, improving code readability and maintainability.

4.4.7.3 Encapsulation and Data Protection:

Encapsulation ensures data protection by hiding the internal implementation details of an object and providing controlled access through methods. It prevents direct manipulation of data, maintaining data integrity and protecting against accidental modification. Encapsulation also allows for easier modifications to the internal implementation without affecting other parts of the code that use the object.

4.4.7.4 Inheritance and Code Reusability:

Inheritance allows the creation of hierarchical relationships between classes, where subclasses inherit properties and behaviors from a superclass. This promotes code reuse and modularity. Inheritance facilitates the creation of specialized classes that inherit common characteristics from a superclass, reducing code duplication and promoting the sharing of common functionality.

4.4.7.5 Polymorphism and Flexibility:

Polymorphism enables objects of different classes to be treated as objects of a common superclass or interface. This flexibility allows for the interchangeability of objects, leading to more modular and extensible code. Polymorphism simplifies

code maintenance and expansion, as new classes can be added without modifying existing code that depends on the common interface.

4.4.7.6 Simplicity and Understandability:

OOP promotes a natural and intuitive way of representing and solving real-world problems by modeling entities as objects with well-defined behaviors and relationships. This approach makes the code easier to understand, maintain, and debug. OOP's emphasis on encapsulation, abstraction, and modularity allows developers to focus on specific components, leading to clearer and more manageable code.

4.4.7.8 Scalability and Maintainability:

OOP provides a structured approach to software development, making it easier to scale and maintain projects. The modular nature of OOP allows for independent development and testing of components, reducing the impact of changes and facilitating code maintenance. The ability to reuse objects and classes also simplifies the addition of new features and enhancements.

4.2 Declaring classes, objects, and methods

Declaring classes, objects, and methods are fundamental aspects of Object-Oriented Programming (OOP).

Declaring Classes: A class is a blueprint or template that defines the properties (attributes) and behaviors (methods) that objects of that class will have. It serves as a blueprint for creating multiple objects with similar characteristics.

Syntax:

```
class ClassName {
    // Class members (fields and methods) }
```

Example:

```
class Person {
    // Class fields
    String name;
    int age;

    // Class methods
    void sayHello() {
        System.out.println("Hello, I'm " + name);
    }

    void celebrateBirthday() {
        age++;
        System.out.println("Happy birthday! I'm now " + age + " years
old.");
    }
}
```

In the example above, the Person class is declared with two fields (name and age) and two methods (sayHello() and celebrateBirthday()). The class serves as a blueprint for creating individual person objects, each with its own name, age, and behavior.

Declaring Objects: An object is an instance of a class. It represents a specific occurrence or realization of the class, with its own unique set of attribute values. Objects have state (values of their attributes) and behavior (methods they can perform).

Syntax:

```
ClassName objectName = new ClassName();
```

Example:

```
Person person1 = new Person();
person1.name = "John";
person1.age = 25;

Person person2 = new Person();
person2.name = "Alice";
person2.age = 30;
```

In the example above, person1 and person2 are two objects created from the Person class. Each object has its own set of attributes (name and age) and can perform the defined methods (sayHello() and celebrateBirthday()).

Declaring Methods: Methods are functions associated with a class. They define the behavior of objects belonging to the class. Methods can access and manipulate the attributes of the class, perform calculations, and interact with other objects or classes.

Syntax:

```
returnType methodName(parameterList) {
    // Method body
    // Statements
}
```

Example:

```
class Calculator {
    int add(int num1, int num2) {
        return num1 + num2;
    }
```

```
    void displayResult(int result) {
        System.out.println("The result is: " + result);
    }
}
```

In the example above, the Calculator class has two methods: add() and displayResult(). The add() method takes two parameters (num1 and num2) and returns their sum. The displayResult() method takes an integer parameter (result) and displays it on the console.

By declaring classes, objects, and methods, we can structure our code into logical units, encapsulate data and behavior, and create reusable components. Objects represent specific instances of a class with their own state, and methods define the operations that objects can perform.

4.3 Constructors and destructors

Constructors and destructors are special methods in object-oriented programming that are used for initializing objects and performing cleanup operations, respectively. Here's an explanation of constructors and destructors with an example:

4.3.1 Constructors

A constructor is a special method that is called automatically when an object is created from a class. It is used to initialize the object's state and set its initial values. Constructors have the same name as the class and do not have a return type.

Syntax:

```
class ClassName {
    // Class fields

    // Constructor
    ClassName() {
        // Constructor body
        // Initialization code
    }
}
```

Example:

```
class Person {
    String name;
    int age;

    // Constructor with parameters
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Default constructor
    Person() {
        name = "Unknown";
        age = 0;
    }
}
```

In the example above, the Person class has two constructors. The first constructor takes two parameters (name and age) and assigns them to the

corresponding fields of the object being created. The second constructor is a default constructor that sets default values for name and age if no values are provided.

Constructors allow for object initialization and provide a way to ensure that objects are in a valid state upon creation. They can be used to assign initial values to instance variables, perform validations, and set up any necessary resources.

4.3.2 Destructors

In Java, there are no explicit destructors like in other languages. Instead, the Java garbage collector automatically handles memory management and cleanup of objects that are no longer referenced. The garbage collector identifies objects that are eligible for garbage collection and reclaims the memory occupied by those objects.

Example:

```
class MyClass {
    // Class members

    // Code executed before object destruction (not a destructor)
    @Override
    protected void finalize() throws Throwable {
        // Cleanup code
        // Resource deallocation
    }
}
```

In the example above, the MyClass class overrides the finalize() method, which is called by the garbage collector before an object is garbage collected. Inside the

finalize() method, cleanup code can be written to deallocate resources or perform any necessary cleanup operations.

It's important to note that the finalize() method is not equivalent to a destructor, as it is not guaranteed to be called immediately or at a specific time. The Java garbage collector determines when to call the finalize() method based on its own algorithms and memory management needs.

constructors are used to initialize objects and ensure they are in a valid state upon creation. They are called automatically when an object is created from a class. Destructors, in the form of the finalize() method, are automatically invoked by the garbage collector before an object is garbage collected, allowing for cleanup operations. Although there are no explicit destructors in Java, the garbage collector handles memory management and cleanup. Constructors and the garbage collector work together to ensure proper object initialization and cleanup in Java.

4.4 Access modifiers (public, private, protected)

Access modifiers in Java are keywords used to specify the accessibility or visibility of classes, methods, and variables. There are three access modifiers: public, private, and protected. Here's an explanation of each access modifier with examples:

4.4.1 Public

The public access modifier allows unrestricted access to a class, method, or variable from any other class or package. Public members are accessible from anywhere, both within the same package and from other packages.

Example:

```
public class MyClass {
    public int publicVar;

    public void publicMethod() {
        // Code
    }
}
```

In the example above, the publicVar and publicMethod() are accessible from anywhere, including other classes and packages. They can be accessed and used without any restrictions.

4.4.2 Private

The private access modifier restricts access to the members of a class only within the same class. Private members are not accessible from other classes or even subclasses. This encapsulates the internal implementation details and prevents direct access to sensitive data.

Example:

```
public class MyClass {
    private int privateVar;

    private void privateMethod() {
        // Code
    }
}
```

In the example above, the privateVar and privateMethod() are accessible only within the MyClass itself. They cannot be accessed from any other class, including subclasses.

4.4.3 Protected

The protected access modifier allows access to the members of a class within the same class, subclasses, and classes within the same package. Protected members are not accessible from classes in other packages.

Example:

```
public class MyClass {
    protected int protectedVar;

    protected void protectedMethod() {
        // Code
    }
}
```

In the example above, the protectedVar and protectedMethod() are accessible within the MyClass, its subclasses, and other classes within the same package. However, they are not accessible from classes in different packages.

Access modifiers provide control over the visibility and accessibility of members in Java, ensuring encapsulation, data hiding, and code security. By choosing the appropriate access modifier, you can define the level of access that other classes and packages have to your code. It's important to use access modifiers judiciously to balance code encapsulation, code maintainability, and code reuse.

4.5 Interfaces and abstract classes

Interfaces define a contract for classes to follow. They specify the methods that implementing classes must implement.

Abstract classes are classes that cannot be instantiated. They serve as a base for other classes and can contain abstract methods.

```
// Interface
public interface Drawable {
    void draw();
}

// Abstract class
public abstract class Shape {
    abstract double area();
}

// Implementing the interface and extending the abstract class
public class Circle extends Shape implements Drawable {
    private double radius;

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

    @Override
    public double area() {
        return Math.PI * radius * radius;
```

```
    }

    @Override
    public void draw() {
        System.out.println("Drawing a circle.");
    }
}

// Creating objects and invoking methods
Circle circle = new Circle(5);
circle.draw();  // Output: "Drawing a circle."
double area = circle.area();
```

Chapter 5: Exception Handling and Error Management

5.1 Understanding exceptions and their types in Java

In Java, exceptions are a mechanism for handling exceptional conditions or errors that occur during program execution. Exceptions allow you to gracefully handle errors and provide a way to separate normal program flow from error-handling logic.

Java exceptions are divided into two categories: checked exceptions and unchecked exceptions.

5.1.1 Checked Exceptions

Checked exceptions are exceptions that must be explicitly declared or caught in the code. If a method can potentially throw a checked exception, it must declare the exception in its method signature using the throws keyword, or handle the exception using a try-catch block. Some common checked exceptions in Java include:

- IOException: This exception is thrown when there is an error during input/output operations, such as reading or writing to a file or network connection.
- SQLException: This exception is thrown when there is an error in database access or SQL operations.
- FileNotFoundException: This exception is thrown when a file being accessed is not found.

5.1.2 Unchecked Exceptions

Unchecked exceptions, also known as runtime exceptions, do not need to be declared or caught explicitly in the code. They are exceptions that occur at runtime and can be handled if necessary. Some common unchecked exceptions in Java include:

- NullPointerException: This exception is thrown when a null reference is accessed.
- ArrayIndexOutOfBoundsException: This exception is thrown when an invalid index is used to access an array.
- IllegalArgumentException: This exception is thrown when an illegal argument is passed to a method.
- ArithmeticException: This exception is thrown when an arithmetic operation fails, such as division by zero.

In addition to the built-in exceptions, you can also create your own custom exceptions by extending the Exception class or one of its subclasses. Custom exceptions allow you to define application-specific exception types that can be thrown and caught in your code.

When an exception is thrown, the program execution is interrupted, and the control is transferred to the nearest appropriate exception handler. This can be done using a try-catch block, where the code that may throw an exception is placed inside the try block, and the exception is caught and handled in the catch block. Finally, the finally block can be used to specify code that will be executed regardless of whether an exception occurs or not.

By properly handling exceptions in your code, you can improve the robustness and reliability of your Java applications.

5.2 The try-catch-finally block and its usage

5.2.1 try block

- The try block is used to enclose a section of code where exceptions may occur.
- It defines the scope within which exceptions are monitored and caught.
- The code inside the try block is executed sequentially, and if an exception occurs, the control is transferred to the appropriate catch block.

5.2.2 catch block

- The catch block is used to handle exceptions that are thrown within the try block.
- It catches and handles specific types of exceptions that occur during program execution.
- Each catch block can handle a specific exception type, allowing for different handling mechanisms based on the type of exception thrown.
- Multiple catch blocks can be used to handle different types of exceptions that might occur.

5.2.3 finally block

- The finally block is used to specify a block of code that is executed regardless of whether an exception occurred or not.
- It follows the try-catch block and is optional, meaning it can be omitted if not needed.
- The finally block is typically used for cleanup activities or releasing resources that were opened in the try block.

- The code inside the finally block is executed even if an exception occurs and is thrown out of the catch block.

Syntax:

```
try {
    // Code that may throw an exception
} catch (ExceptionType1 exception1) {
    // Exception handling for ExceptionType1
} catch (ExceptionType2 exception2) {
    // Exception handling for ExceptionType2
} finally {
    // Code that executes regardless of exceptions
}
```

Example:

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class FileReadExample {
    public static void main(String[] args) {
        FileInputStream inputStream = null;
        try {
            inputStream = new FileInputStream("myfile.txt");
            int data;
            while ((data = inputStream.read()) != -1) {
                System.out.print((char) data);
            }
```

```
        } catch (FileNotFoundException e) {
            System.err.println("File not found: " + e.getMessage());
        } catch (IOException e) {
            System.err.println("Error reading file: " +
e.getMessage());
        } finally {
            try {
                if (inputStream != null) {
                    inputStream.close();
                }
            } catch (IOException e) {
                System.err.println("Error closing file: " +
e.getMessage());
            }
            System.out.println("Program execution completed.");
        }
    }
}
```

In the above example, the FileReadExample class attempts to read the contents of a file using a FileInputStream within the try block. If the file is not found, a FileNotFoundException is caught and handled in the corresponding catch block. If an error occurs during file reading, an IOException is caught and handled in another catch block. The finally block is used to close the file input stream, releasing the associated resources, and to print a completion message.

The try-catch-finally block provides a structured approach to handle exceptions and ensure proper resource management. It allows you to handle exceptions gracefully, preventing unexpected program termination and providing meaningful feedback to users. The finally block guarantees that critical cleanup tasks are

performed, even in the presence of exceptions, promoting stability and reliability in Java applications.

5.3 Throwing and propagating exceptions

Throwing and propagating exceptions in Java allows you to handle exceptional situations by signaling that an error or unexpected condition has occurred. It enables you to raise exceptions explicitly and pass them up the call stack for handling at higher levels. Here's an explanation of throwing and propagating exceptions with an example:

5.3.1 Throwing Exceptions

- To throw an exception, you use the throw keyword followed by an instance of an exception class or a subclass thereof.
- Throwing an exception allows you to indicate that an error or exceptional condition has occurred within a method.
- You can throw exceptions explicitly based on specific conditions or requirements.

Syntax:

```
throw new ExceptionType("Error message");
```

Example:

```
public class DivisionExample {
    public static int divide(int dividend, int divisor) throws
ArithmeticException {
        if (divisor == 0) {
```

```
            throw new ArithmeticException("Division by zero is not
allowed.");
        }
        return dividend / divisor;
    }

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

In the above example, the divide() method takes two parameters (dividend and divisor) and performs integer division. If the divisor is zero, an ArithmeticException is explicitly thrown using the throw keyword. The exception message indicates the specific error that occurred.

5.3.2 Propagating Exceptions

- Propagating exceptions involves throwing an exception from a method and allowing it to be handled by the calling method or propagated further up the call stack.
- When a method throws an exception, it is required to declare the exception using the throws clause in its method signature.
- The calling method or the higher-level caller can choose to handle the exception or propagate it further.

Syntax:

```
public returnType methodName(parameters) throws ExceptionType1,
ExceptionType2 {
    // Method body
    // May throw exceptions
}
```

Example:

```
public class FileReaderExample {
    public static void readFile(String filename) throws
FileNotFoundException {
        FileInputStream inputStream = new FileInputStream(filename);
        // Code to read file
        // May throw IOException
    }

    public static void main(String[] args) {
        try {
            readFile("myfile.txt");
        } catch (FileNotFoundException e) {
            System.err.println("File not found: " + e.getMessage());
        }
    }
}
```

In the above example, the readFile() method throws a FileNotFoundException because it attempts to open and read a file. The method signature includes the throws FileNotFoundException clause, indicating that it can potentially throw this exception. In the main() method, the exception is caught and handled using a catch block.

By throwing and propagating exceptions, you can handle exceptional situations explicitly and communicate errors effectively. Throwing exceptions allows you to signal and handle errors within methods, while propagating exceptions enables higher-level code to handle exceptions appropriately. This helps in achieving error handling, fault tolerance, and maintainability in Java applications.

Chapter 6: Java Collections and Generics

6.1 Overview of Java collections framework

The Java Collections Framework is a set of classes and interfaces that provide a standardized way of managing and manipulating groups of objects. It offers a wide range of data structures and algorithms to store, retrieve, and process collections of objects efficiently. Here's an overview of the Java Collections Framework and its usage with an example:

6.1.1 Core Interfaces

The Java Collections Framework includes several core interfaces that define the behavior and functionality of different types of collections. Some of the key interfaces are:

- Collection: Represents a generic collection of objects.
- List: Represents an ordered collection that allows duplicate elements.
- Set: Represents an unordered collection that does not allow duplicate elements.
- Map: Represents a mapping between unique keys and values.

6.1.2 Common Implementations

The framework provides various implementations of the core interfaces to suit different needs. Some commonly used implementations include:

- ArrayList: Implements the List interface using a dynamic array.
- LinkedList: Implements the List interface using a doubly-linked list.
- HashSet: Implements the Set interface using a hash table.
- TreeSet: Implements the Set interface using a balanced binary tree.
- HashMap: Implements the Map interface using a hash table.
- TreeMap: Implements the Map interface using a balanced binary tree.

6.1.3 Usage and Benefits:

- Collections provide a more flexible and efficient way to store and manage groups of objects compared to arrays.
- The Java Collections Framework allows you to perform various operations on collections, such as adding, removing, and retrieving elements.
- Collections offer built-in algorithms and methods for sorting, searching, and manipulating data.
- The framework provides improved memory utilization and performance optimizations for different types of collections.
- It supports generics, enabling type safety and compile-time type checking.
- Collections can be easily integrated with other Java features like streams, lambda expressions, and functional programming constructs.

Example:

```
import java.util.ArrayList;
import java.util.List;

public class CollectionExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");
```

```
        System.out.println("Names: " + names);

        if (names.contains("Bob")) {
            System.out.println("Bob is in the list.");
        }

        names.remove("Charlie");

        System.out.println("Size: " + names.size());

        for (String name : names) {
            System.out.println("Hello, " + name + "!");
        }
    }
}
```

In the example above, an ArrayList is created to store a collection of names. Names are added to the list using the add() method. The contains() method is used to check if the list contains the name "Bob". The remove() method removes the name "Charlie" from the list. The size() method returns the size of the list. Finally, a loop is used to iterate over the elements of the list and print a greeting message for each name.

The Java Collections Framework provides a rich set of data structures and algorithms to handle collections of objects efficiently. It simplifies the management and manipulation of groups of data, offering increased productivity and code reusability. By leveraging the framework's interfaces and implementations, you can achieve optimal performance and maintainable code in your Java applications.

6.2 Working with collections: adding, accessing, removing elements

Working with collections in Java involves adding, accessing, and removing elements from a collection. The Java Collections Framework provides various methods to perform these operations efficiently. Here's a detailed explanation of adding, accessing, and removing elements from collections, along with an example:

6.2.1 Adding Elements

- To add elements to a collection, you can use the add() method provided by the collection's interface.
- The add() method adds the specified element to the collection at the end (or at a specific index, in some cases).
- Depending on the type of collection, duplicate elements may or may not be allowed.

Example:

```
import java.util.ArrayList;
import java.util.List;

public class AddElementsExample {
    public static void main(String[] args) {
        List<String> colors = new ArrayList<>();
        colors.add("Red");
        colors.add("Green");
        colors.add("Blue");
        System.out.println("Colors: " + colors);
    }
}
```

In the above example, an ArrayList called colors is created to store a collection of color names. The add() method is used to add three color names to the list. The output displays the contents of the colors list.

6.2.2 Accessing Elements

- To access elements from a collection, you can use various methods provided by the collection's interface, such as get(), iterator(), or enhanced for-loop.
- The get() method retrieves the element at the specified index.
- The iterator allows you to iterate over the collection and access each element sequentially.
- The enhanced for-loop simplifies the process of iterating over the collection.

Example:

```
import java.util.ArrayList;
import java.util.List;

public class AccessElementsExample {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Orange");

        System.out.println("First fruit: " + fruits.get(0));

        System.out.println("All fruits:");
        for (String fruit : fruits) {
```

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

In the above example, an ArrayList called fruits is created to store a collection of fruit names. The add() method is used to add three fruit names to the list. The get() method is used to access and print the first fruit in the list. The enhanced for-loop is used to iterate over the collection and print all the fruits.

6.2.3 Removing Elements

- To remove elements from a collection, you can use methods like remove(), removeAll(), or clear().
- The remove() method removes the specified element from the collection.
- The removeAll() method removes all the elements from the collection that are present in another collection.
- The clear() method removes all the elements from the collection, making it empty.

Example:

```
import java.util.ArrayList;
import java.util.List;

public class RemoveElementsExample {
    public static void main(String[] args) {
        List<String> cities = new ArrayList<>();
        cities.add("New York");
        cities.add("London");
        cities.add("Paris");
```

```
        System.out.println("Cities: " + cities);

        cities.remove("London");
        System.out.println("After removing 'London': " + cities);

        cities.clear();
        System.out.println("After clearing the list: " + cities);
    }
}
```

In the above example, an ArrayList called cities is created to store a collection of city names. The add() method is used to add three city names to the list. The remove() method is used to remove the city "London" from the list. The clear() method is used to remove all the elements from the list, making it empty.

By understanding how to add, access, and remove elements from collections in Java, you can effectively manage and manipulate data in your applications. The Java Collections Framework provides a rich set of methods and interfaces to perform these operations efficiently, allowing you to work with collections seamlessly.

6.3 Iterating through collections using iterators and enhanced for loop

Iterators allow you to traverse through collections and perform actions on each element.

The Iterator interface provides methods like hasNext() and next() to iterate through elements.

The enhanced for loop (for-each loop) simplifies the process of iterating over collections.

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

public class IterationExample {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();

        names.add("John");
        names.add("Jane");
        names.add("Mike");

        // Using iterator
        Iterator<String> iterator = names.iterator();
        while (iterator.hasNext()) {
            String name = iterator.next();
            System.out.println(name);
        }
```

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

6.4 Understanding generics and type safety in Java

Generics provide a way to specify the type of objects that a collection can hold.
Generics ensure type safety by preventing incompatible types from being added to a collection.
Generics allow you to write reusable and type-safe code.

6.5 Using generic classes, interfaces, and methods

Generic classes and interfaces can be defined with type parameters that represent the types of objects they work with.

Generic methods can have their own type parameters independent of the class or interface.

Generic types can be specified during instantiation or method invocation.

```
public class GenericExample<T> {
    private T value;
```

```
    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }

    public static <E> void printArray(E[] array) {
        for (E element : array) {
            System.out.println(element);
        }
    }
}

// Using generic class
GenericExample<String> stringExample = new GenericExample<>();
stringExample.setValue("Hello");
String stringValue = stringExample.getValue();

// Using generic method
Integer[] numbers = {1, 2, 3, 4, 5};
GenericExample.printArray(numbers);
```

By understanding the Java collectionsframework, working with collections, iterating through them using iterators and enhanced for loops, grasping the concept of generics and type safety, and utilizing generic classes, interfaces, and methods, you can effectively manage and manipulate groups of objects

in a type-safe and efficient manner. This enables you to create flexible and reusable code that adapts to different data types and provides better code readability and maintainability.

Chapter 7: Multithreading and Concurrent Programming

7.1 Introduction to multithreading and its benefits

Multithreading is a programming concept that allows multiple threads of execution to run concurrently within a single program. It enables a program to perform multiple tasks simultaneously, improving efficiency and responsiveness. Here's an introduction to multithreading and its benefits, along with an example and explanation:

7.1.1 Multithreading Concept:

A thread is a lightweight unit of execution within a program. It represents an independent flow of control.
Multithreading involves dividing a program into smaller units of execution called threads, which can run concurrently.
Each thread operates independently, performing its own set of instructions and accessing shared resources.
Multithreading can be used to perform multiple tasks simultaneously, maximize CPU utilization, and improve overall program performance.

7.1.2 Benefits of Multithreading:

Concurrent Execution: Multithreading enables concurrent execution of multiple tasks, allowing a program to make progress on multiple fronts simultaneously.

Responsiveness: Multithreading enhances the responsiveness of applications by allowing background tasks to run independently without blocking the user interface.

Resource Utilization: By utilizing multiple threads, a program can make better use of available CPU resources, improving overall system performance.
Modularity and Maintainability: Multithreading allows developers to organize code into separate threads, enhancing modularity and making code maintenance easier.

Asynchronous Operations: Multithreading enables asynchronous operations, where one thread can perform time-consuming tasks while other threads continue execution, leading to more efficient utilization of resources.

Example:

```
public class MultiThreadExample {
    public static void main(String[] args) {
        // Create and start two threads
        Thread thread1 = new Thread(new Task("Thread 1"));
        Thread thread2 = new Thread(new Task("Thread 2"));
        thread1.start();
        thread2.start();

        // Perform main thread tasks
        for (int i = 0; i < 5; i++) {
            System.out.println("Main thread executing...");
            try {
                Thread.sleep(1000); // Pause for 1 second
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
        }
    }

    // A simple task that prints a message
    static class Task implements Runnable {
        private String name;

        public Task(String name) {
            this.name = name;
        }

        public void run() {
            for (int i = 0; i < 5; i++) {
                System.out.println("Task " + name + " executing...");
                try {
                    Thread.sleep(1000); // Pause for 1 second
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

In the above example, the MultiThreadExample class demonstrates multithreading. Two threads, thread1 and thread2, are created and started using the Thread class. Each thread executes the Task class, which implements the Runnable interface. The Task class performs a simple task of printing a message

multiple times with a pause of 1 second between each iteration. Meanwhile, the main thread also executes its own set of tasks. The concurrent execution of threads can be observed in the console output.

By utilizing multithreading, you can leverage the power of concurrency and improve the performance and responsiveness of your Java applications. It allows for parallel execution, efficient resource utilization, and the ability to handle multiple tasks simultaneously. However, it's important to ensure proper synchronization and thread safety to avoid issues like data races and thread interference.

7.2 Creating and running threads in Java

In Java, threads can be created by extending the Thread class or implementing the Runnable interface.

To run a thread, you invoke the start() method, which calls the run() method internally.

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

// Creating and running a thread
MyThread thread = new MyThread();
thread.start();
```

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

// Creating and running a thread using Runnable
Thread thread = new Thread(new MyRunnable());
thread.start();
```

7.3 Synchronization and thread safety

Synchronization is the process of coordinating the execution of multiple threads to ensure thread safety.

Thread safety refers to the property of a program that behaves correctly when executed by multiple threads concurrently.

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

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

// Creating multiple threads that increment the counter
Counter counter = new Counter();
```

```
Runnable incrementTask = () -> {
    for (int i = 0; i < 1000; i++) {
        counter.increment();
    }
};

Thread thread1 = new Thread(incrementTask);
Thread thread2 = new Thread(incrementTask);

thread1.start();
thread2.start();

thread1.join();
thread2.join();

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

Chapter 8: Networking and Client-Server Communication

8.1 Overview of networking in Java

Networking in Java allows you to develop applications that can communicate over a network using various protocols such as TCP/IP, UDP, HTTP, and more. Java provides a comprehensive set of classes and APIs for networking, making it easier to create networked applications. Here's an overview of networking in Java with an example and explanation:

8.1.1 Key Classes and Interfaces

java.net.Socket: Represents a client-side endpoint for communication with a server. It provides methods to connect to a server, send and receive data.

java.net.ServerSocket: Represents a server-side endpoint that listens for incoming client connections on a specific port.

java.net.InetAddress: Represents an IP address or a domain name. It provides methods to resolve hostnames and obtain IP addresses.

java.net.URL: Represents a Uniform Resource Locator and provides methods to access resources on the internet.

java.net.HttpURLConnection: Provides methods for making HTTP requests to web servers and handling responses.

8.1.2 Client-Server Communication

- In a client-server model, the client initiates a connection to a server and sends requests, while the server listens for incoming connections and responds to client requests.
- The client uses the Socket class to connect to the server's IP address and port number. It can then send and receive data over the socket.
- The server uses the ServerSocket class to listen for incoming client connections. When a client connects, the server creates a Socket object to communicate with the client.

Example - Client-Server Communication:

```
// Server code
import java.io.*;
import java.net.*;

public class Server {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(8080)) {
            System.out.println("Server started. Listening on port
8080...");

            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("Client connected: " +
clientSocket.getInetAddress());

                // Process client request
                // ...
```

```
                clientSocket.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

// Client code
import java.io.*;
import java.net.*;

public class Client {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 8080)) {
            System.out.println("Connected to server.");

            // Send request to server
            OutputStream outputStream = socket.getOutputStream();
            PrintWriter writer = new PrintWriter(outputStream, true);
            writer.println("GET / HTTP/1.1");
            writer.println("Host: localhost");
            writer.println();

            // Receive response from server
            InputStream inputStream = socket.getInputStream();
            BufferedReader reader = new BufferedReader(new
InputStreamReader(inputStream));
            String line;
            while ((line = reader.readLine()) != null) {
```

```
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

In the above example, the server code listens for incoming client connections on port 8080 using a ServerSocket. When a client connects, it creates a Socket object to communicate with the client. The server can process client requests as needed. The client code connects to the server's IP address and port using a Socket. It sends an HTTP GET request to the server and reads the response.

Networking in Java enables you to build various networked applications, such as client-server applications, web crawlers, chat applications, and more. With the available classes and APIs, you can easily establish network connections, send and receive data, and work with different network protocols. It provides a powerful foundation for developing robust and scalable networked applications in Java.

8.2 Understanding TCP/IP and UDP protocols

TCP/IP and UDP are two widely used protocols in networking. They operate at the transport layer of the TCP/IP protocol suite and serve different purposes. Here's an explanation of TCP/IP and UDP protocols along with an example to illustrate their differences:

8.2.1 TCP/IP (Transmission Control Protocol/Internet Protocol):

- TCP/IP is a reliable, connection-oriented protocol that provides error checking, sequencing, and flow control.
- It guarantees the delivery of data packets in the correct order without loss or duplication.
- TCP establishes a connection between a client and a server before data transfer and ensures that data is received accurately.
- It is suitable for applications that require reliable and ordered delivery of data, such as file transfer, email, and web browsing.

8.2.2 UDP (User Datagram Protocol):

- UDP is a lightweight, connectionless protocol that offers fast, unreliable delivery of data packets.
- It does not guarantee the delivery of packets and does not perform error checking, sequencing, or flow control.
- UDP is used in applications where speed and low latency are more important than reliability, such as real-time streaming, online gaming, and DNS.
- It is ideal for scenarios where occasional packet loss is acceptable, and real-time communication is crucial.

Example:

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

public class TCPExample {
    public static void main(String[] args) {
        String serverName = "localhost";
```

```
        int port = 8080;

        try {
            // TCP client
            Socket clientSocket = new Socket(serverName, port);

            // Send data to the server
            OutputStream outputStream =
clientSocket.getOutputStream();
            PrintWriter writer = new PrintWriter(outputStream, true);
            writer.println("Hello, Server!");

            // Receive data from the server
            InputStream inputStream = clientSocket.getInputStream();
            BufferedReader reader = new BufferedReader(new
InputStreamReader(inputStream));
            String response = reader.readLine();
            System.out.println("Server response: " + response);

            // Close the connection
            clientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

public class UDPExample {
    public static void main(String[] args) {
        String serverName = "localhost";
        int port = 8080;
```

```
        try {
            // UDP client
            DatagramSocket clientSocket = new DatagramSocket();

            // Send data to the server
            String message = "Hello, Server!";
            byte[] sendData = message.getBytes();
            InetAddress serverAddress =
InetAddress.getByName(serverName);
            DatagramPacket sendPacket = new DatagramPacket(sendData,
sendData.length, serverAddress, port);
            clientSocket.send(sendPacket);

            // Receive data from the server
            byte[] receiveData = new byte[1024];
            DatagramPacket receivePacket = new
DatagramPacket(receiveData, receiveData.length);
            clientSocket.receive(receivePacket);
            String response = new String(receivePacket.getData(), 0,
receivePacket.getLength());
            System.out.println("Server response: " + response);

            // Close the socket
            clientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

In the above example, the TCP client and UDP client demonstrate the differences between TCP/IP and UDP protocols. Both clients send a "Hello, Server!" message to the server and receive a response.

- The TCP client establishes a connection using a Socket, sends the message over the connection, and reads the response using streams (InputStream and OutputStream).
- The UDP client creates a DatagramSocket, constructs a DatagramPacket with the message and server address, sends the packet, and receives the response using datagrams (DatagramPacket and DatagramSocket).

The TCP client guarantees the reliable delivery of the message, while the UDP client does not provide any guarantee and relies on the best-effort delivery model.

Understanding the differences between TCP/IP and UDP protocols allows you to choose the appropriate protocol based on the requirements of your networked application. TCP is suitable for applications that prioritize reliability, while UDP is more appropriate for applications that require low latency and real-time communication.

8.3 Building client-server applications using sockets

Building client-server applications using sockets involves establishing a connection between a client and a server to enable communication. The client and server exchange data through network sockets, allowing them to send and receive information. Here's a detailed explanation of building client-server applications using sockets, along with an example and explanation:

8.3.1 Server-Side Implementation:

- The server-side implementation involves creating a server socket, listening for incoming client connections, and handling client requests.
- The server creates a ServerSocket object, binds it to a specific port, and waits for incoming client connections using the accept() method.
- Once a client connection is accepted, the server creates a separate thread to handle the client's request concurrently.
- The server reads the client's input, processes it, and sends back a response.

Example:

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

public class Server {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(8080)) {
            System.out.println("Server started. Listening on port
8080...");

            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("Client connected: " +
clientSocket.getInetAddress());

                // Create a new thread to handle client requests
                Thread thread = new Thread(new
ClientHandler(clientSocket));
                thread.start();
```

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

class ClientHandler implements Runnable {
    private Socket clientSocket;

    public ClientHandler(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }

    @Override
    public void run() {
        try {
            // Get input/output streams from the client socket
            BufferedReader reader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
            PrintWriter writer = new
PrintWriter(clientSocket.getOutputStream(), true);

            // Read client input
            String clientInput = reader.readLine();
            System.out.println("Client input: " + clientInput);

            // Process client request
            String response = "Hello, Client!";
            writer.println(response);
            System.out.println("Response sent: " + response);
```

```
            // Close client socket
            clientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

In the above example, the Server class represents the server-side implementation. It listens on port 8080 using a ServerSocket. Once a client connection is accepted, a new thread (ClientHandler) is created to handle the client request. The ClientHandler class implements the Runnable interface and performs the necessary operations to read the client's input, process it, and send a response.

8.3.2 Client-Side Implementation

- The client-side implementation involves creating a client socket, connecting it to the server's IP address and port, and sending/receiving data.
- The client creates a Socket object, providing the server's IP address and port number.
- It obtains input/output streams from the socket to read from and write to the server.
- The client can send data to the server using the output stream and receive the server's response using the input stream.

Example:

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

```
public class Client {
    public static void main(String[] args) {
        String serverName = "localhost";
        int port = 8080;

        try (Socket socket = new Socket(serverName, port)) {
            System.out.println("Connected to server.");

            // Get input/output streams from the socket
            BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter writer = new
PrintWriter(socket.getOutputStream(), true);

            // Send data to the server
            writer.println("Hello, Server!");

            // Receive data from the server
            String serverResponse = reader.readLine();
            System.out.println("Server response: " + serverResponse);

            // Close the socket
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

In the above example, the Client class represents the client-side implementation. It creates a Socket object, providing the server's IP address and port. It obtains the input/output streams from the socket to communicate with the server. The client sends the message "Hello, Server!" to the server using the output stream and reads the server's response from the input stream.

By building client-server applications using sockets, you can establish communication between different devices over a network. The server listens for client connections, handles client requests, and sends responses. The client connects to the server, sends data, and receives the server's response. This allows for seamless communication and interaction between clients and servers in various networked applications.

8.4 Sending and receiving data over the network

Data can be sent and received over a network using input and output streams.

The InputStream and OutputStream classes provide basic functionality for reading and writing data.

```
// Server-side code
try (ServerSocket serverSocket = new ServerSocket(8080)) {
    Socket clientSocket = serverSocket.accept();

    // Receiving data from client
    BufferedReader reader = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
    String clientMessage = reader.readLine();
    System.out.println("Received from client: " + clientMessage);
```

```
    // Sending data to client
    PrintWriter writer = new
PrintWriter(clientSocket.getOutputStream(), true);
    writer.println("Message received!");

    clientSocket.close();
} catch (IOException e) {
    e.printStackTrace();
}

// Client-side code
try (Socket socket = new Socket("localhost", 8080)) {
    // Sending data to server
    PrintWriter writer = new PrintWriter(socket.getOutputStream(),
true);
    writer.println("Hello from client!");

    // Receiving data from server
    BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
    String serverResponse = reader.readLine();
    System.out.println("Received from server: " + serverResponse);

    socket.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

8.5 Error handling and robust network communication

Error handling in network communication involves handling exceptions and ensuring graceful recovery from network-related issues.

Robust network communication includes handling timeouts, handling disconnections, and implementing retry mechanisms.

```
// Server-side code
try (ServerSocket serverSocket = new ServerSocket(8080)) {
    serverSocket.setSoTimeout(5000); // Set a timeout of 5 seconds

    try {
        Socket clientSocket = serverSocket.accept();
        // Communication code here
    } catch (SocketTimeoutException e) {
        System.out.println("Timeout occurred. No client connected
within the specified time.");
    }

    serverSocket.close();
} catch (IOException e) {
    e.printStackTrace();
}

// Client-side code
try (Socket socket = new Socket("localhost", 8080)) {
    socket.setSoTimeout(5000); // Set a timeout of 5 seconds

    // Communication code here
```

```
    socket.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

By understanding networking in Java, TCP/IP and UDP protocols, building client-server applications using sockets, sending and receiving data over the network, and implementing error handling and robust network communication strategies, you can develop robust and reliable networked applications that facilitate communication between clients and servers.

#JAVA PROJECTS

15 JAVA PROJECTS

CODING HUBS

Index

Chapter 9: Projects

9.1 Stopwatch

```
public class Stopwatch {
    private long startTime;
    private long stopTime;
    private boolean running;

    public void start() {
        if (!running) {
            startTime = System.currentTimeMillis();
            running = true;
            System.out.println("Stopwatch started.");
        } else {
            System.out.println("Stopwatch is already running.");
        }
    }

    public void stop() {
        if (running) {
            stopTime = System.currentTimeMillis();
            running = false;
            System.out.println("Stopwatch stopped.");
        } else {
            System.out.println("Stopwatch is not running.");
        }
    }

    public long getElapsedTime() {
        if (running) {
            return System.currentTimeMillis() - startTime;
        } else {
            return stopTime - startTime;
        }
    }

    public void reset() {
```

```
        startTime = 0;
        stopTime = 0;
        running = false;
        System.out.println("Stopwatch reset.");
    }

    public static void main(String[] args) {
        Stopwatch stopwatch = new Stopwatch();

        stopwatch.start();

        // Simulating some process...
        try {
            Thread.sleep(2000); // Sleep for 2 seconds
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        stopwatch.stop();

        System.out.println("Elapsed time: " +
stopwatch.getElapsedTime() + " milliseconds");

        stopwatch.reset();
    }
}
```

9.2 Calendar Application

```
import java.util.Calendar;
public class CalendarApplication {
   public static void main(String[] args) {
   Calendar calendar = Calendar.getInstance();
   System.out.println("The current date is : " + calendar.getTime());
   calendar.add(Calendar.DATE, -15);
   System.out.println("15 days ago: " + calendar.getTime());
   calendar.add(Calendar.MONTH, 4);
   System.out.println("4 months later: " + calendar.getTime());
   calendar.add(Calendar.YEAR, 2);
```

```
   System.out.println("2 years later: " + calendar.getTime());
   }
}
```

9.3 Address Book

```
import java.util.Scanner;

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

        //create a table to hold information

        String[][] addressbooks = new String[100][8];

        addressbooks[0][0]="Mobile Number";
        addressbooks[0][1]="First Name";
        addressbooks[0][2]="Last Name";
        addressbooks[0][3]="Address";
        addressbooks[0][4]="City";
        addressbooks[0][5]="County";
        addressbooks[0][7]="Telephone Number";

        //pre-populate address book for testing purposes and records

        addressbooks[1][0]="1";
        addressbooks[1][1]="David";
        addressbooks[1][2]="Needham";
        addressbooks[1][3]="Sraheens, Achill";
        addressbooks[1][4]="Galway";
        addressbooks[1][5]="Mayo";
        addressbooks[1][6]="086-1581077";
        addressbooks[1][7]="098-45368";

        addressbooks[2][0]="2";
        addressbooks[2][1]="Mc";
```

```
        addressbooks[2][2]="lovin";
        addressbooks[2][3]="Hawaii";
        addressbooks[2][4]="Hawaii";
        addressbooks[2][5]="Hawaii";
        addressbooks[2][6]="12345";
        addressbooks[2][7]="412-555-1234";

        //menu options
        System.out.print("Welcome to my Address book!");
        System.out.print("\n");
        System.out.print("\n1 - Insert a New Contact \n2 - Search
Contact by Last Name \n3 - Delete Contact \n4 - Show All Contacts \n5
- Exit " );
        System.out.print("\n");
        System.out.print("\nChoose your option: ");

        int option = input.nextInt();

        if (option ==1)
        {
            System.out.print("\nPlease enter your First Name : ");
        }
        if (option ==2)
        {
        }

        if (option ==3)
        {
        }

        if (option ==4)
        {
        System.out.println(addressbooks[1][0]+
        "\t"+addressbooks[1][2]+ ", "+addressbooks[1][1]+
        "\n\t"+addressbooks[1][3]+
        "\n\t"+addressbooks[1][4]+ ", "+addressbooks[1][5]+ "
"+addressbooks[1][6]+
        "\n\t"+addressbooks[1][7]);
        }
```

```
        if (option ==5)
        {
        }

    }
}
```

9.4 Simple Paint Application

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Collection;
import java.util.HashSet;
import java.util.Set;

import javax.imageio.ImageIO;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

class SimplePaintPanel extends JPanel {
   private final Set<Point> blackPixels = new HashSet<Point>();
   private final int brushSize;
```

```
private int mouseButtonDown = 0;

public SimplePaintPanel() {
   this(5, new HashSet<Point>());
}/*from   w  w  w  .j  a  va2s.  c om*/

public SimplePaintPanel(Set<Point> blackPixels) {
   this(5, blackPixels);
}

public SimplePaintPanel(int brushSize, Set<Point> blackPixels) {
   this.setPreferredSize(new Dimension(300, 300));
   this.brushSize = brushSize;
   this.blackPixels.addAll(blackPixels);
   final SimplePaintPanel self = this;

   MouseAdapter mouseAdapter = new MouseAdapter() {
      @Override
      public void mouseDragged(MouseEvent ev) {
         if (mouseButtonDown == 1)
            addPixels(getPixelsAround(ev.getPoint()));
         else if (mouseButtonDown == 3)
            removePixels(getPixelsAround(ev.getPoint()));
      }

      @Override
      public void mousePressed(MouseEvent ev) {
         self.mouseButtonDown = ev.getButton();
      }
   };
   this.addMouseMotionListener(mouseAdapter);
   this.addMouseListener(mouseAdapter);

}

public void paint(Graphics g) {
   int w = this.getWidth();
   int h = this.getHeight();
   g.setColor(Color.white);
   g.fillRect(0, 0, w, h);
```

```
      g.setColor(Color.black);
      for (Point point : blackPixels)
         g.drawRect(point.x, point.y, 1, 1);

   }

   public void clear() {
      this.blackPixels.clear();
      this.invalidate();
      this.repaint();
   }

   public void addPixels(Collection<? extends Point> blackPixels) {
      this.blackPixels.addAll(blackPixels);
      this.invalidate();
      this.repaint();
   }

   public void removePixels(Collection<? extends Point> blackPixels) {
      this.blackPixels.removeAll(blackPixels);
      this.invalidate();
      this.repaint();
   }

   public boolean isPixel(Point blackPixel) {
      return this.blackPixels.contains(blackPixel);
   }

   private Collection<? extends Point> getPixelsAround(Point point) {
      Set<Point> points = new HashSet<>();
      for (int x = point.x - brushSize; x < point.x + brushSize; x++)
         for (int y = point.y - brushSize; y < point.y + brushSize;
y++)
            points.add(new Point(x, y));
      return points;
   }
}

public class Main extends JFrame implements ActionListener {
   private final String ACTION_NEW = "New Image";
```

```
private final String ACTION_LOAD = "Load Image";
private final String ACTION_SAVE = "Save Image";

private final SimplePaintPanel paintPanel = new SimplePaintPanel();

public Main() {
   super();
   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   setTitle("Simple Paint");

   initMenu();
   this.getContentPane().add(paintPanel);

   pack();
   setVisible(true);
}

private void initMenu() {
   JMenuBar menuBar = new JMenuBar();
   JMenu menu = new JMenu("File");
   JMenuItem mnuNew = new JMenuItem(ACTION_NEW);
   JMenuItem mnuLoad = new JMenuItem(ACTION_LOAD);
   JMenuItem mnuSave = new JMenuItem(ACTION_SAVE);
   mnuNew.setActionCommand(ACTION_NEW);
   mnuLoad.setActionCommand(ACTION_LOAD);
   mnuSave.setActionCommand(ACTION_SAVE);
   mnuNew.addActionListener(this);
   mnuLoad.addActionListener(this);
   mnuSave.addActionListener(this);
   menu.add(mnuNew);
   menu.add(mnuLoad);
   menu.add(mnuSave);
   menuBar.add(menu);
   this.setJMenuBar(menuBar);
}

@Override
public void actionPerformed(ActionEvent ev) {
   switch (ev.getActionCommand()) {
   case ACTION_NEW:
```

```
            paintPanel.clear();
            break;
         case ACTION_LOAD:
            doLoadImage();
            break;
         case ACTION_SAVE:
            doSaveImage();
            break;
         }
      }

      private void doSaveImage() {
         JFileChooser fileChooser = new JFileChooser();
         fileChooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
         int result = fileChooser.showSaveDialog(this);
         if (result != JFileChooser.APPROVE_OPTION)
            return;
         File saveFile = fileChooser.getSelectedFile();
         if (!saveFile.getAbsolutePath().toLowerCase().endsWith(".png"))
            saveFile = new File(saveFile.getAbsolutePath() + ".png");
         BufferedImage image = new
   BufferedImage(paintPanel.getSize().width, paintPanel.getSize().height,
               BufferedImage.TYPE_INT_RGB);
         for (int x = 0; x < image.getWidth(); x++) {
            for (int y = 0; y < image.getHeight(); y++) {
               image.setRGB(x, y, Color.white.getRGB());
               if (paintPanel.isPixel(new Point(x, y))) {
                  image.setRGB(x, y, Color.black.getRGB());
               }
            }
         }
         try {
            ImageIO.write(image, "png", saveFile);
         } catch (IOException e) {
            return;
         }
      }

      private void doLoadImage() {
         JFileChooser fileChooser = new JFileChooser();
```

```
        fileChooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
        int result = fileChooser.showOpenDialog(this);
        if (result != JFileChooser.APPROVE_OPTION)
           return;
        BufferedImage image;
        File openFile = fileChooser.getSelectedFile();
        try (FileInputStream fis = new FileInputStream(openFile)) {
           image = ImageIO.read(fis);
        } catch (IOException e) {
           return;
        }
        if (image == null)
           return;
        paintPanel.clear();
        Set<Point> blackPixels = new HashSet<Point>();
        for (int x = 0; x < image.getWidth(); x++) {
           for (int y = 0; y < image.getHeight(); y++) {
              Color c = new Color(image.getRGB(x, y));
              if ((c.getBlue() < 128 || c.getRed() < 128 || c.getGreen()
< 128) && c.getAlpha() == 255) {
                 blackPixels.add(new Point(x, y));
              }
           }
        }
        paintPanel.addPixels(blackPixels);
     }

     public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
           public void run() {
              new Main();
           }
        });
     }
}
```

9.5 Password Generator

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

```
import java.util.Scanner;

public class generator {
    // function to generate password
    static String generate_password(int size) {
        // collection of characters that can be used in password
        String chars =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890!@#$%^&
*()_+-/.,<>?;':\"[]{}\\|`~";

        String password = "";
        // creating object of Random class
        Random rnd = new Random();
        // looping to generate password
        while (password.length() < size) {
            // get a random number between 0 and length of chars
            int index = (int) (rnd.nextFloat() * chars.length());
            // add character at index to password
            password += chars.charAt(index);
        }
        return password;
    }

    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the size of password: ");
        int size = sc.nextInt();
        sc.close();

        // calling function to generate password
        String password = generate_password(size);
        // printing the password
        System.out.println(password);
    }
}
```

9.6 URL Shortener

```
// Java program to generate short url from integer id and
// integer id back from short url.
import java.util.*;
import java.lang.*;
import java.io.*;

class GFG
{
      // Function to generate a short url from integer ID
      static String idToShortURL(int n)
      {
            // Map to store 62 possible characters
            char map[] =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789".toCha
rArray();

            StringBuffer shorturl = new StringBuffer();

            // Convert given integer id to a base 62 number
            while (n > 0)
            {
                  // use above map to store actual character
                  // in short url
                  shorturl.append(map[n % 62]);
                  n = n / 62;
            }

            // Reverse shortURL to complete base conversion
            return shorturl.reverse().toString();
      }

      // Function to get integer ID back from a short url
      static int shortURLtoID(String shortURL)
      {
            int id = 0; // initialize result
```

```
            // A simple base conversion logic
            for (int i = 0; i < shortURL.length(); i++)
            {
                  if ('a' <= shortURL.charAt(i) &&
                              shortURL.charAt(i) <= 'z')
                  id = id * 62 + shortURL.charAt(i) - 'a';
                  if ('A' <= shortURL.charAt(i) &&
                              shortURL.charAt(i) <= 'Z')
                  id = id * 62 + shortURL.charAt(i) - 'A' + 26;
                  if ('0' <= shortURL.charAt(i) &&
                              shortURL.charAt(i) <= '9')
                  id = id * 62 + shortURL.charAt(i) - '0' + 52;
            }
            return id;
      }

      // Driver Code
      public static void main (String[] args) throws IOException
      {
            int n = 12345;
            String shorturl = idToShortURL(n);
            System.out.println("Generated short url is " + shorturl);
            System.out.println("Id from url is " +
                                          shortURLtoID(shorturl));
      }
}
```

9.7 File Compression/Decompression

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.zip.DeflaterOutputStream;
public class CompressingFiles {
   public static void main(String args[]) throws IOException {
      //Instantiating the FileInputStream
      String inputPath = "D:\ExampleDirectory\logo.jpg";
      FileInputStream inputStream = new FileInputStream(inputPath);
      //Instantiating the FileOutputStream
```

```
      String outputPath = "D:\ExampleDirectory\compressedLogo.txt";
      FileOutputStream outputStream = new
FileOutputStream(outputPath);
      //Instantiating the DeflaterOutputStream
      DeflaterOutputStream compresser = new
DeflaterOutputStream(outputStream);
      int contents;
      while ((contents=inputStream.read())!=-1){
         compresser.write(contents);
      }
      compresser.close();
      System.out.println("File compressed.......");
   }
}
```

9.8 Scientific Calculator

```
import java.util.Scanner;

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

        System.out.println("Scientific Calculator");
        System.out.println("1. Addition");
        System.out.println("2. Subtraction");
        System.out.println("3. Multiplication");
        System.out.println("4. Division");
        System.out.println("5. Square Root");
        System.out.println("6. Power");

        System.out.print("Enter your choice: ");
        int choice = scanner.nextInt();

        double result;

        switch (choice) {
            case 1:
                System.out.print("Enter the first number: ");
```

```
                double num1 = scanner.nextDouble();
                System.out.print("Enter the second number: ");
                double num2 = scanner.nextDouble();
                result = num1 + num2;
                System.out.println("Result: " + result);
                break;
            case 2:
                System.out.print("Enter the first number: ");
                num1 = scanner.nextDouble();
                System.out.print("Enter the second number: ");
                num2 = scanner.nextDouble();
                result = num1 - num2;
                System.out.println("Result: " + result);
                break;
            case 3:
                System.out.print("Enter the first number: ");
                num1 = scanner.nextDouble();
                System.out.print("Enter the second number: ");
                num2 = scanner.nextDouble();
                result = num1 * num2;
                System.out.println("Result: " + result);
                break;
            case 4:
                System.out.print("Enter the dividend: ");
                num1 = scanner.nextDouble();
                System.out.print("Enter the divisor: ");
                num2 = scanner.nextDouble();
                if (num2 != 0) {
                    result = num1 / num2;
                    System.out.println("Result: " + result);
                } else {
                    System.out.println("Error: Division by zero is not
allowed.");
                }
                break;
            case 5:
                System.out.print("Enter a number: ");
                num1 = scanner.nextDouble();
                if (num1 >= 0) {
                    result = Math.sqrt(num1);
```

```
                    System.out.println("Square root: " + result);
                } else {
                    System.out.println("Error: Cannot calculate square
root of a negative number.");
                }
                break;
            case 6:
                System.out.print("Enter the base: ");
                num1 = scanner.nextDouble();
                System.out.print("Enter the exponent: ");
                num2 = scanner.nextDouble();
                result = Math.pow(num1, num2);
                System.out.println("Result: " + result);
                break;
            default:
                System.out.println("Invalid choice!");
                break;
        }

        scanner.close();
    }
}
```

9.9 Phonebook Application

```
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

public class PhonebookApplication {
    private Map<String, String> phonebook;

    public PhonebookApplication() {
        phonebook = new HashMap<>();
    }

    public void addContact(String name, String phoneNumber) {
        phonebook.put(name, phoneNumber);
        System.out.println("Contact added: " + name);
```

```
    }

    public void searchContact(String name) {
        if (phonebook.containsKey(name)) {
            String phoneNumber = phonebook.get(name);
            System.out.println("Name: " + name + ", Phone Number: " +
phoneNumber);
        } else {
            System.out.println("Contact not found: " + name);
        }
    }

    public void deleteContact(String name) {
        if (phonebook.containsKey(name)) {
            phonebook.remove(name);
            System.out.println("Contact deleted: " + name);
        } else {
            System.out.println("Contact not found: " + name);
        }
    }

    public void displayAllContacts() {
        System.out.println("Phonebook Contacts:");
        for (Map.Entry<String, String> entry : phonebook.entrySet()) {
            System.out.println("Name: " + entry.getKey() + ", Phone
Number: " + entry.getValue());
        }
    }

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

        while (true) {
            System.out.println("Phonebook Application");
            System.out.println("1. Add Contact");
            System.out.println("2. Search Contact");
            System.out.println("3. Delete Contact");
            System.out.println("4. Display All Contacts");
```

```
            System.out.println("5. Exit");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();

            switch (choice) {
                case 1:
                    System.out.print("Enter the name: ");
                    scanner.nextLine(); // Consume the newline
character
                    String name = scanner.nextLine();
                    System.out.print("Enter the phone number: ");
                    String phoneNumber = scanner.nextLine();
                    phonebookApp.addContact(name, phoneNumber);
                    break;
                case 2:
                    System.out.print("Enter the name: ");
                    scanner.nextLine(); // Consume the newline
character
                    name = scanner.nextLine();
                    phonebookApp.searchContact(name);
                    break;
                case 3:
                    System.out.print("Enter the name: ");
                    scanner.nextLine(); // Consume the newline
character
                    name = scanner.nextLine();
                    phonebookApp.deleteContact(name);
                    break;
                case 4:
                    phonebookApp.displayAllContacts();
                    break;
                case 5:
                    System.out.println("Exiting Phonebook
Application...");
                    scanner.close();
                    System.exit(0);
                default:
                    System.out.println("Invalid choice! Please try
again.");
                    break;
```

```
            }
        }
    }
}
```

9.10 BMI Calculator

```
import java.util.Scanner;

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

        System.out.println("BMI Calculator");
        System.out.print("Enter your weight in kilograms: ");
        double weight = scanner.nextDouble();

        System.out.print("Enter your height in meters: ");
        double height = scanner.nextDouble();

        double bmi = calculateBMI(weight, height);
        String bmiCategory = getBMICategory(bmi);

        System.out.println("Your BMI: " + bmi);
        System.out.println("Category: " + bmiCategory);

        scanner.close();
    }

    public static double calculateBMI(double weight, double height) {
        return weight / (height * height);
    }

    public static String getBMICategory(double bmi) {
        if (bmi < 18.5) {
            return "Underweight";
        } else if (bmi < 25) {
            return "Normal weight";
        } else if (bmi < 30) {
```

```
            return "Overweight";
        } else {
            return "Obese";
        }
    }
}
```

9.11 Alarm Clock

```
import java.time.LocalTime;
import java.util.Scanner;

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

        System.out.println("Alarm Clock");
        System.out.print("Enter the alarm time in HH:MM format (24-
hour clock): ");
        String alarmTimeInput = scanner.nextLine();

        // Parse the alarm time input
        String[] parts = alarmTimeInput.split(":");
        int alarmHour = Integer.parseInt(parts[0]);
        int alarmMinute = Integer.parseInt(parts[1]);

        // Create the LocalTime object for the alarm time
        LocalTime alarmTime = LocalTime.of(alarmHour, alarmMinute);

        // Get the current time
        LocalTime currentTime = LocalTime.now();

        // Calculate the time difference
        long timeDifference = calculateTimeDifference(currentTime,
alarmTime);

        if (timeDifference > 0) {
            System.out.println("Alarm set successfully!");
            try {
```

```
                // Sleep for the remaining time
                Thread.sleep(timeDifference);
                System.out.println("Wake up! It's time!");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        } else {
            System.out.println("Invalid alarm time. Please enter a
future time.");
        }

        scanner.close();
    }

    public static long calculateTimeDifference(LocalTime currentTime,
LocalTime alarmTime) {
        int currentHour = currentTime.getHour();
        int currentMinute = currentTime.getMinute();
        int currentSecond = currentTime.getSecond();

        int alarmHour = alarmTime.getHour();
        int alarmMinute = alarmTime.getMinute();
        int alarmSecond = alarmTime.getSecond();

        // Convert the current time and alarm time to seconds
        long currentTimeInSeconds = (currentHour * 3600) +
(currentMinute * 60) + currentSecond;
        long alarmTimeInSeconds = (alarmHour * 3600) + (alarmMinute *
60) + alarmSecond;

        // Calculate the time difference
        long timeDifference = alarmTimeInSeconds -
currentTimeInSeconds;

        // Adjust the time difference if it's negative (alarm time is
earlier than current time)
        if (timeDifference < 0) {
            timeDifference += 24 * 3600; // Add 24 hours (in seconds)
to the time difference
        }
```

```
        return timeDifference * 1000; // Convert seconds to
milliseconds
    }
}
```

9.12 Random Quote Generator

```
import java.util.Random;

public class RandomQuoteGenerator {
    private static final String[] quotes = {
        "Be yourself; everyone else is already taken. - Oscar Wilde",
        "Two things are infinite: the universe and human stupidity;
and I'm not sure about the universe. - Albert Einstein",
        "You only live once, but if you do it right, once is enough. -
Mae West",
        "The only way to do great work is to love what you do. - Steve
Jobs",
        "In three words I can sum up everything I've learned about
life: it goes on. - Robert Frost",
        "The best revenge is massive success. - Frank Sinatra",
        "The future belongs to those who believe in the beauty of
their dreams. - Eleanor Roosevelt",
        "Life is what happens when you're busy making other plans. -
John Lennon",
        "Don't count the days, make the days count. - Muhammad Ali",
        "The greatest glory in living lies not in never falling, but
in rising every time we fall. - Nelson Mandela"
    };

    public static void main(String[] args) {
        Random random = new Random();
        int randomIndex = random.nextInt(quotes.length);
        String randomQuote = quotes[randomIndex];
        System.out.println("Random Quote:");
        System.out.println(randomQuote);
    }
```

```
}
```

9.13 Personal Expense Tracker

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class ExpenseTracker {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        List<Expense> expenses = new ArrayList<>();

        System.out.println("Personal Expense Tracker");

        while (true) {
            System.out.println("1. Add Expense");
            System.out.println("2. View Expenses");
            System.out.println("3. Exit");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();

            switch (choice) {
                case 1:
                    addExpense(scanner, expenses);
                    break;
                case 2:
                    viewExpenses(expenses);
                    break;
                case 3:
                    System.out.println("Exiting Expense Tracker...");
                    scanner.close();
                    System.exit(0);
                default:
                    System.out.println("Invalid choice! Please try
again.");
                    break;
            }
        }
```

```
    }

    public static void addExpense(Scanner scanner, List<Expense>
expenses) {
        scanner.nextLine(); // Consume the newline character

        System.out.print("Enter the description: ");
        String description = scanner.nextLine();

        System.out.print("Enter the amount: ");
        double amount = scanner.nextDouble();

        Expense expense = new Expense(description, amount);
        expenses.add(expense);

        System.out.println("Expense added successfully!");
    }

    public static void viewExpenses(List<Expense> expenses) {
        if (expenses.isEmpty()) {
            System.out.println("No expenses found.");
        } else {
            System.out.println("Expense List:");
            for (Expense expense : expenses) {
                System.out.println(expense);
            }
        }
    }
}

class Expense {
    private String description;
    private double amount;

    public Expense(String description, double amount) {
        this.description = description;
        this.amount = amount;
    }

    @Override
```

```
    public String toString() {
        return "Description: " + description + ", Amount: " + amount;
    }
}
```

9.14 Weather App

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class WeatherApp {
    public static void main(String[] args) {
        try {
            // Specify the URL of the weather API
            String weatherUrl =
"https://api.openweathermap.org/data/2.5/weather?q=London&appid=YOUR_A
PI_KEY";

            // Send an HTTP GET request to the API
            HttpURLConnection connection = (HttpURLConnection) new
URL(weatherUrl).openConnection();
            connection.setRequestMethod("GET");

            // Get the response from the API
            int responseCode = connection.getResponseCode();

            if (responseCode == HttpURLConnection.HTTP_OK) {
                // Read the response
                BufferedReader reader = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
                StringBuilder response = new StringBuilder();
                String line;

                while ((line = reader.readLine()) != null) {
                    response.append(line);
```

```
                }

                reader.close();

                // Parse the response as needed
                String weatherData = response.toString();
                System.out.println("Weather Data: " + weatherData);
            } else {
                System.out.println("Error: Failed to retrieve weather
data. Response Code: " + responseCode);
            }
        } catch (IOException e) {
            System.out.println("Error: Failed to connect to the
weather API.");
            e.printStackTrace();
        }
    }
}
```

9.15 To-Do List Application

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class ToDoList {
    private List<String> tasks;

    public ToDoList() {
        tasks = new ArrayList<>();
    }

    public void addTask(String task) {
        tasks.add(task);
        System.out.println("Task added: " + task);
    }

    public void removeTask(int index) {
        if (index >= 0 && index < tasks.size()) {
```

```
            String removedTask = tasks.remove(index);
            System.out.println("Task removed: " + removedTask);
        } else {
            System.out.println("Invalid task index.");
        }
    }

    public void displayTasks() {
        if (tasks.isEmpty()) {
            System.out.println("No tasks found.");
        } else {
            System.out.println("Task List:");
            for (int i = 0; i < tasks.size(); i++) {
                System.out.println((i + 1) + ". " + tasks.get(i));
            }
        }
    }

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

        while (true) {
            System.out.println("To-Do List Application");
            System.out.println("1. Add Task");
            System.out.println("2. Remove Task");
            System.out.println("3. Display Tasks");
            System.out.println("4. Exit");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();

            switch (choice) {
                case 1:
                    scanner.nextLine(); // Consume the newline
character
                    System.out.print("Enter the task: ");
                    String task = scanner.nextLine();
                    toDoList.addTask(task);
                    break;
                case 2:
```

```
                    System.out.print("Enter the task index: ");
                    int index = scanner.nextInt();
                    toDoList.removeTask(index - 1);
                    break;
                case 3:
                    toDoList.displayTasks();
                    break;
                case 4:
                    System.out.println("Exiting To-Do List
Application...");
                    scanner.close();
                    System.exit(0);
                default:
                    System.out.println("Invalid choice! Please try
again.");
                    break;
            }
        }
    }
}
```

9.16 Currency Exchange Calculator

```
import java.text.DecimalFormat;
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

public class CurrencyExchangeCalculator {
    private static final Map<String, Double> exchangeRates = new
HashMap<>();

    static {
        // Initialize exchange rates
        exchangeRates.put("USD", 1.0);
        exchangeRates.put("EUR", 0.85);
        exchangeRates.put("GBP", 0.73);
        exchangeRates.put("JPY", 109.86);
    }
```

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

        System.out.println("Currency Exchange Calculator");

        System.out.print("Enter the amount: ");
        double amount = scanner.nextDouble();

        System.out.print("Enter the source currency code: ");
        String sourceCurrency = scanner.next().toUpperCase();

        System.out.print("Enter the target currency code: ");
        String targetCurrency = scanner.next().toUpperCase();

        double convertedAmount = convertCurrency(amount,
sourceCurrency, targetCurrency);

        DecimalFormat decimalFormat = new DecimalFormat("#.##");
        String formattedAmount =
decimalFormat.format(convertedAmount);

        System.out.println(amount + " " + sourceCurrency + " = " +
formattedAmount + " " + targetCurrency);

        scanner.close();
    }

    public static double convertCurrency(double amount, String
sourceCurrency, String targetCurrency) {
        if (exchangeRates.containsKey(sourceCurrency) &&
exchangeRates.containsKey(targetCurrency)) {
            double sourceRate = exchangeRates.get(sourceCurrency);
            double targetRate = exchangeRates.get(targetCurrency);
            return amount * (targetRate / sourceRate);
        } else {
            throw new IllegalArgumentException("Invalid currency
code.");
        }
    }
```

```
}
```