Java Programming for Beginners
DCODE Club, JEC

Instructor: Firoz, Forheen, Zubayer

4th November, 2023

Contents

1 Introduction to Java

Java is a popular and versatile programming language known for its platform independence. In this section, we will provide a detailed introduction to Java, covering important topics for beginners.

1.1 What is Java?

Java is a high-level, object-oriented programming language developed by Sun Microsystems (now owned by Oracle). It was first released in 1995 and has since become one of the most widely used languages for software development. Here are some key features of Java:

- **Platform Independence:** Java programs can run on any platform (Windows, macOS, Linux, etc.) with a compatible Java Virtual Machine (JVM).
- **Object-Oriented:** Java follows the object-oriented programming paradigm, which emphasizes the use of classes and objects to model real-world entities.
- **Strongly Typed:** Java is a strongly typed language, which means that variables have specific data types, and type compatibility is enforced.
- **Automatic Memory Management:** Java includes a garbage collector that automatically manages memory, reducing the risk of memory leaks.
- **Rich Standard Library:** Java offers a vast standard library, providing pre-built classes and methods for common tasks.
- **Security:** Java is known for its robust security features, making it suitable for developing secure applications.

1.2 Why Learn Java?

Learning Java can open up numerous opportunities in the world of software development. Here are some compelling reasons to learn Java as a beginner:

- **Versatility:** Java is used in various domains, including web development, mobile app development (Android), server-side programming, and more.
- **High Demand:** Java developers are in high demand in the job market, making it a valuable skill.
- **Community and Resources:** Java has a large and active developer community, resulting in abundant online resources and support.
- **Career Opportunities:** Java programmers can pursue careers as software developers, web developers, mobile app developers, and more.
- **Learning Path:** Java provides a structured learning path for beginners to understand programming concepts, making it an ideal choice for newcomers to coding.

1.3 Setting up the Java Development Environment

Before you start writing Java code, you need to set up your development environment. Here are the key steps:

1. **Install the Java Development Kit (JDK):** The JDK includes the Java compiler (javac) and the Java Virtual Machine (JVM). Download and install the appropriate version of the JDK for your operating system from the official Oracle website.
2. **Install an Integrated Development Environment (IDE):** While it's possible to write Java code using a simple text editor, using an IDE can significantly improve your productivity. Popular Java IDEs include Eclipse, IntelliJ IDEA, and NetBeans.
3. **Write Your First Java Program:** After setting up your environment, write a simple "Hello, World!" program to ensure everything is working correctly. This is a traditional first step when learning any programming language.

Once you have your development environment set up, you're ready to dive into Java programming. In the following sections, we will cover fundamental concepts and gradually build your programming skills.

2 Java Code Compilation and Platform Independence

Java, a popular programming language, is known for its platform independence. This attribute is achieved through a combination of the Java Development Kit (JDK), the Java Runtime Environment (JRE), and the Java Virtual Machine (JVM). In this section, we'll explore how Java code is compiled and executed, and why it can run on various platforms.

2.1 Java Code Compilation VS Other Popular Languages

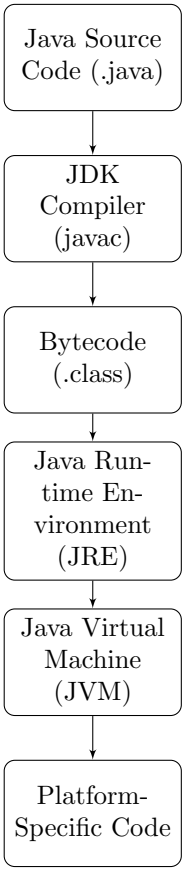

Figure 1: Java Compilation and Execution

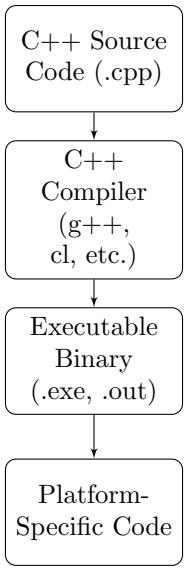

Figure 2: C++ Compilation and Execution

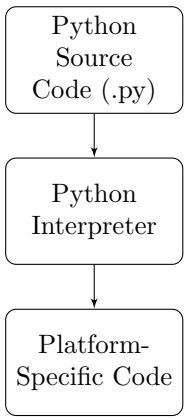

Figure 3: Python Execution (No Compilation)

2.2 Java Platform Independence

The platform independence of Java is a result of several key features:

- **Bytecode:** Java code is compiled to bytecode, which is a low-level representation of the program. This bytecode is platform-independent and can be executed on any system that has a compatible JRE.
- **JVM (Java Virtual Machine):** The JVM is responsible for executing Java bytecode. It abstracts the underlying hardware and provides a consistent execution environment. The JVM interprets bytecode and adapts it to the host system, allowing the same program to run on different platforms.
- **Class Libraries:** Java includes a vast standard library, the Java Standard Library (Java API), which is available on all platforms. This ensures that fundamental functionality is consistent across different systems.
- **Write Once, Run Anywhere (WORA):** The WORA principle encapsulates the idea that Java developers can write their code once and run it on various platforms without modification. This is possible due to the compilation to bytecode and the JVM's role in providing platform abstraction.

Variable Type	Size (in bytes)
`byte`	1
`short`	2
`int`	4
`long`	8
`float`	4
`double`	8
`char`	2
`boolean`	1 (typically)
`Object`	4 (reference)
`String`	Variable (depends on content)

Table 1: Size of Java Variable Types

In conclusion, the Java ecosystem, consisting of the JDK, JRE, and JVM, allows Java code to be compiled into platform-independent bytecode, which can be executed on any system with a compatible JRE. This powerful combination of technology is what makes Java a versatile and platform-independent programming language.

3 Basic Concepts

In this section, we will delve into the fundamental concepts of Java programming that every beginner should understand. These concepts are essential for building a solid foundation in Java.

3.1 Variables and Data Types

3.1.1 What are Variables?

In Java, a variable is a container that holds data. It has a name and a data type. Variables allow you to store and manipulate data within your programs. Here are some key points about variables:

- **Declaration:** Variables must be declared before they can be used. The declaration specifies the variable's name and data type.
- **Data Types:** Java has several data types, including `int`, `double`, `char`, and more. Each data type determines the kind of data a variable can hold.
- **Initialization:** After declaration, variables can be assigned initial values. Uninitialized variables contain a default value.
- **Naming Rules:** Variable names are case-sensitive and must begin with a letter, underscore (_), or dollar sign ($). They can contain letters, digits, underscores, and dollar signs.

3.1.2 Common Data Types

Java provides various data types to represent different kinds of data. Here are some common data types:

- **int:** Used for storing integer values.
- **double:** Used for storing floating-point (decimal) numbers.
- **char:** Used for single characters, such as letters and symbols.
- **boolean:** Used for storing true or false values.

3.1.3 Examples

Let's look at some examples of variable declarations and assignments:

```
int age;            // Declaration of an integer variable
age = 25;           // Assignment of a value to the 'age' variable

double price = 19.99; // Declaration and initialization of a double variable

char grade = 'A';    // Declaration and initialization of a char variable

boolean isJavaFun = true; // Declaration and initialization of a boolean variable
```

3.2 Operators

Operators are symbols that perform operations on variables and values. Java provides various types of operators, including:

- **Arithmetic Operators:** Used for mathematical calculations, such as addition (+), subtraction (-), multiplication (*), division (/), and modulus (%).
- **Relational Operators:** Used to compare values, returning a boolean result. Examples include equal to (==), not equal to (!=), greater than (¿), and less than (¡).
- **Logical Operators:** Used to perform logical operations on boolean values. Common logical operators are AND (&&), OR (||), and NOT (!).
- **Assignment Operators:** Used to assign values to variables with shorthand notation, such as += and -=.
- **Increment and Decrement Operators:** Used to increase or decrease the value of a variable by 1, with variations like ++ and –.

3.2.1 Examples

Here are some examples of using operators in Java:

```
int x = 10;
int y = 5;

int sum = x + y;          // Addition
int difference = x - y;  // Subtraction
int product = x * y;     // Multiplication
int quotient = x / y;    // Division
int remainder = x % y;    // Modulus

boolean isGreater = x > y;              // Relational operator
boolean logicalAnd = (x > 0) && (y > 0); // Logical AND
boolean logicalOr = (x > 0) || (y > 0);  // Logical OR

x++;  // Increment by 1
y--;  // Decrement by 1
```

Listing 1: Java Operators Example

3.3 Input and Output

Java programs often need to interact with the user by reading input and displaying output. This can be accomplished using various methods and classes.

3.3.1 Reading Input

To read input from the user, you can use the Scanner class, which is available in the java.util package. Here's how you can read input from the keyboard: import java.util.Scanner;

Scanner scanner = new Scanner(System.in); System.out.print("Enter your name: "); String name = scanner.nextLine(); System.out.println("Hello, " + name + "!");

3.3.2 Displaying Output

You can display output in Java using the System.out.println method. It prints text to the console and moves the cursor to the next line. System.out.println("This is a message."); You can also use the System.out.print method to print text without moving to the next line. System.out.print("This is "); System.out.print("on the same line."); In this section, we've covered the fundamental concepts of variables, data types, operators, and input/output. These concepts are essential for writing and understanding Java programs. In the next section, we will explore control structures, which allow you to make decisions and control the flow of your programs.

4 Control Structures

Control structures in Java allow you to make decisions and control the flow of your programs. In this section, we will explore two fundamental types of control structures: conditional statements and loops.

4.1 Conditional Statements

Conditional statements allow you to execute different blocks of code based on specified conditions. In Java, you have three primary conditional statements:

- **if Statement:** The 'if' statement allows you to execute a block of code if a specified condition is true. Optionally, you can include an 'else' block to handle the case where the condition is false.
- **else-if Statement:** The 'else-if' statement allows you to check multiple conditions in a sequence and execute the code block associated with the first true condition.
- **switch Statement:** The 'switch' statement provides an efficient way to test a single variable against multiple possible values. It's commonly used when you have a specific set of values to compare.

4.1.1 if Statement

The 'if' statement is a fundamental control structure. It allows you to conditionally execute code based on a true or false condition. Here's the basic syntax:

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

4.1.2 else-if Statement

The 'else-if' statement allows you to test multiple conditions in a sequence. It's useful when you have more than two possible outcomes. Here's the syntax:

```
if (condition1) {
    // Code to execute if condition1 is true
} else if (condition2) {
    // Code to execute if condition2 is true
} else {
    // Code to execute if none of the conditions are true
}
```

4.1.3 switch Statement

The ‘switch‘ statement is used to select one of many code blocks to be executed. It’s especially useful when you need to test a single variable against multiple values. Here’s the syntax:

```
switch (variable) {
    case value1:
        // Code to execute if variable is equal to value1
        break;
    case value2:
        // Code to execute if variable is equal to value2
        break;
    // ... More cases ...
    default:
        // Code to execute if none of the cases match
}
```

4.1.4 Ternary Operator

The ternary operator is a concise way to express conditional logic in a single line. It is often used for simple conditional assignments. Here’s the syntax:

```
result = (condition) ? valueIfTrue : valueIfFalse;
```

The ‘condition‘ is evaluated, and if it’s true, ‘valueIfTrue‘ is assigned to ‘result‘; otherwise, ‘valueIfFalse‘ is assigned to ‘result‘.

4.2 Loops

Loops in Java allow you to repeatedly execute a block of code. Java provides several types of loops, including:

- **for Loop:** The ‘for‘ loop is used for executing a block of code a specified number of times. It’s commonly used when you know the number of iterations in advance.
- **while Loop:** The ‘while‘ loop repeatedly executes a block of code as long as a specified condition is true. It’s often used when you don’t know the number of iterations beforehand.
- **do-while Loop:** The ‘do-while‘ loop is similar to the ‘while‘ loop but guarantees that the code block is executed at least once before checking the condition.

4.2.1 for Loop

The ‘for‘ loop is used to execute a block of code a specified number of times. It consists of an initialization, condition, and increment (or decrement) expression. Here’s the syntax:

```
for (initialization; condition; increment) {
    // Code to execute in each iteration
}
```

4.2.2 while Loop

The ‘while‘ loop repeatedly executes a block of code as long as a specified condition is true. Here’s the syntax:

```
while (condition) {
    // Code to execute in each iteration
}
```

4.2.3 do-while Loop

The 'do-while' loop is similar to the 'while' loop but guarantees that the code block is executed at least once before checking the condition. Here's the syntax:

```
do {
    // Code to execute in each iteration
} while (condition);
```

In this section, we have explored control structures, including conditional statements (if, else-if, switch) and loops (for, while, do-while). These control structures are essential for building dynamic and responsive programs in Java.

5 Time and Space Complexity

Time and space complexity analysis is crucial for evaluating the efficiency of algorithms. It allows us to estimate how the runtime and memory usage of an algorithm scale with input size. In this section, we will discuss the notation used to represent complexity and provide examples.

5.1 Time Complexity

Time complexity measures the amount of time an algorithm takes to complete based on the input size. The most common notations are:

- $O(\cdot)$: Big O notation represents the upper bound or worst-case time complexity. It describes the upper limit of the algorithm's runtime.
- $\Omega(\cdot)$: Omega notation represents the lower bound or best-case time complexity. It describes the lower limit of the algorithm's runtime.
- $\Theta(\cdot)$: Theta notation represents the tight bound or average-case time complexity. It provides a more precise estimate of the algorithm's runtime.

Time Complexity	Description
$\Theta(1)$	Constant Time
$\Theta(\log n)$	Logarithmic Time
$\Theta(n)$	Linear Time
$\Theta(n \log n)$	Linearithmic Time
$\Theta(n^2)$	Quadratic Time
$\Theta(n^3)$	Cubic Time
$\Theta(2^n)$	Exponential Time
$\Theta(n!)$	Factorial Time

Table 2: Common Time Complexities

5.1.1 Example: Linear Time Complexity

Consider a simple linear search algorithm:

```
public static int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target) {
            return i; // Element found at index i
        }
    }
    return -1; // Element not found
}
```

The time complexity of linear search is $O(n)$ in the worst case, where n is the size of the array. This means that the algorithm's runtime grows linearly with the input size.

5.2 Space Complexity

Space complexity measures the amount of memory an algorithm uses based on the input size. The space complexity is represented with the same notations as time complexity: $O(\cdot)$, $\Omega(\cdot)$, and $\Theta(\cdot)$.

5.2.1 Example: Constant Space Complexity

Consider a function that calculates the sum of two integers:

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

The space complexity of this function is $O(1)$, regardless of the values of a and b. It uses a constant amount of memory to store the result, and the memory usage does not depend on the input values.

5.3 Summary

Understanding time and space complexity helps in choosing the right algorithm for specific tasks. It allows us to analyze and predict the performance of algorithms under different scenarios and input sizes. In the following sections, we will explore various sorting and searching algorithms and discuss their time and space complexities in more detail.

6 Functions and Methods

Functions and methods in Java are blocks of code that can be defined, called, and reused. They play a vital role in breaking down complex tasks into smaller, manageable pieces. In this section, we will explore how to define, call, and use functions and methods in Java.

6.1 Defining Functions

In Java, functions are typically called methods, and they are defined within classes. A method is a block of code that performs a specific task and is identified by a unique name. Methods can take input parameters (arguments) and return a value.

Here's the basic syntax for defining a method in Java:

```
returnType methodName(parameterType1 parameterName1, parameterType2
   parameterName2, ...) {
    // Method code
    return result; // Optional return statement
}
```

- 'returnType': Specifies the data type of the value that the method will return. Use 'void' if the method does not return any value. - 'methodName': The name of the method, which you choose. - 'parameterType1', 'parameterType2', ...: Data types of the method's parameters. - 'parameterName1', 'parameterName2', ...: Names of the method's parameters.

6.2 Calling Functions

To use a method in Java, you need to call it by its name and provide the required arguments. Here's how you call a method:

```
returnType result = methodName(argument1, argument2, ...);
```

- 'returnType': If the method returns a value, you can capture it in a variable of the specified data type. - 'methodName': The name of the method you want to call. - 'argument1', 'argument2', ...: Values or expressions that match the method's parameters.

6.3 Example: A Simple Method

Let's define and use a simple method that calculates the sum of two numbers.

"'java int add(int a, int b) return a + b;

public static void main(String[] args) int num1 = 5; int num2 = 7; int sum = add(num1, num2); System.out.println("Sum: " + sum); In this example, the add method takes two integers as parameters, calculates their sum, and returns the result. In the main method, we call the add method and print the result.

6.4 Method Overloading

Java allows you to define multiple methods with the same name but different parameter lists. This is called method overloading. When you call an overloaded method, Java determines which version to execute based on the provided arguments.

6.5 Built-in Methods

Java provides a wide range of built-in methods that you can use for various tasks. For example, you can use System.out.println() to print to the console, Math.sqrt() to calculate the square root, and many more. These methods are part of Java's standard library and can be used without defining them.

6.6 User-Defined Methods

Apart from built-in methods, you can define your own methods to perform custom tasks. User-defined methods are essential for organizing your code, improving reusability, and making your programs more readable.

In this section, we have covered the basics of defining, calling, and using functions and methods in Java. Methods are a crucial part of Java programming, enabling you to create modular and maintainable code. You can continue to explore more advanced topics related to methods as you become more proficient in Java development.

7 Arrays and Strings

Arrays and strings are essential data structures in Java, allowing you to store and manipulate collections of data. In this section, we will explore how to work with arrays and strings.

7.1 Arrays

An array is a data structure that holds a fixed-size collection of elements of the same data type. Each element in an array is accessed by its index, starting from 0. Here's how you declare an array in Java:

```
dataType[] arrayName = new dataType[arraySize];
```

- `dataType`: The data type of the elements in the array. - `arrayName`: The name you choose for the array. - `arraySize`: The number of elements the array can hold.

7.1.1 Initializing Arrays

You can initialize an array with values when declaring it:

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

7.1.2 Accessing Elements

To access elements in an array, use the square brackets and the index:

```
int firstNumber = numbers[0]; // Access the first element
```

7.1.3 Iterating Through Arrays

You can use loops to iterate through array elements, making it easy to perform operations on all elements:

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

7.1.4 2D Arrays

In Java, you can create multidimensional arrays, including 2D arrays. A 2D array is like a table or grid, where elements are organized in rows and columns. Here's how you declare and initialize a 2D array:

```
dataType[][] twoDArray = new dataType[rows][columns];
```

- `dataType`: The data type of the elements in the array. - `twoDArray`: The name you choose for the 2D array. - `rows`: The number of rows in the array. - `columns`: The number of columns in the array.

Accessing and iterating through 2D arrays involve using nested loops, one for the rows and another for the columns.

7.1.5 How Arrays Are Stored in Memory

In Java, arrays are stored as contiguous blocks of memory. The elements are arranged in memory one after the other, making it efficient for direct access based on index. The memory allocation for arrays ensures that each element can be accessed in constant time.

For 2D arrays, memory is allocated sequentially for rows and columns. Elements are stored row by row, and accessing an element requires knowing both the row and column indices.

Understanding the memory organization of arrays is important for optimizing memory usage and efficient data retrieval in Java.

7.2 Strings

A string in Java is a sequence of characters. Strings are objects, and Java provides a rich set of methods to manipulate and work with them.

7.2.1 String Declaration and Initialization

You can declare and initialize strings in several ways:

```
String greeting = "Hello, World!";
String name = new String("Alice");
```

7.2.2 String Concatenation

You can concatenate strings using the '+' operator:

```
String firstName = "John";
String lastName = "Doe";
String fullName = firstName + " " + lastName;
```

7.2.3 String Methods

Java provides numerous methods to work with strings, such as:

- `length()`: Returns the length of the string. - `charAt(index)`: Returns the character at the specified index. - `substring(start, end)`: Returns a substring between the start and end positions. - `equals(otherString)`: Compares two strings for equality.

7.2.4 StringBuilder for Efficient String Manipulation

While string concatenation using the '+' operator works, it can be inefficient for extensive string manipulation. Java provides the 'StringBuilder' class for more efficient string handling:

```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.append("Hello");
stringBuilder.append(" ");
stringBuilder.append("World");
String result = stringBuilder.toString(); // Convert StringBuilder to a String
```

7.2.5 Char Data Type

Java has a data type called 'char' to represent a single character. It is commonly used for individual characters within strings:

```
char grade = 'A';
char symbol = '$';
```

7.2.6 Memory Consumption of Strings

In Java, strings are immutable, which means once a string is created, it cannot be changed. Whenever you modify a string (e.g., by concatenation or substring operations), a new string is created in memory. This behavior has implications for memory consumption and performance.

Here are some key points to understand about memory consumption when working with strings:

1. **Immutable Strings:** When you create a string, it remains fixed and cannot be modified. If you need to make changes, such as appending characters, a new string object is created. This can lead to a proliferation of string objects in memory.

2. **String Concatenation:** Using the '+' operator for string concatenation creates new string objects. For example, when you concatenate two strings, a new string with the combined contents is created.

3. **String Interning:** Java performs string interning, which means it can reuse string objects with the same content. However, interning is not a solution to excessive memory consumption, as not all strings are interned.

4. **Memory Overhead:** Each string object in Java carries additional memory overhead for bookkeeping, which includes the length of the string, character data, and other metadata.

5. **StringBuilder for Efficiency:** To mitigate memory consumption when you need to perform numerous string modifications, use the 'StringBuilder' class. 'StringBuilder' is a mutable alternative to strings and is more memory-efficient for frequent string manipulations.

6. **StringBuffer:** Another option for efficient string manipulation is the 'StringBuffer' class, which is similar to 'StringBuilder' but is synchronized and thread-safe. Use 'StringBuffer' when working with strings in multi-threaded environments.

7. **String Pool:** Java maintains a string pool, which is a pool of unique strings in memory. It allows some degree of reusing strings with the same content. However, it's important to note that not all strings are automatically added to the pool.

Understanding how strings are managed in terms of memory is crucial for efficient Java programming, especially in applications where memory efficiency is essential.

8 Object-Oriented Programming in Java

8.1 Introduction to Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a fundamental programming paradigm used in Java. It is based on the concept of "objects," which represent real-world entities and their interactions. OOP promotes code organization and reusability by modeling the program as a collection of objects that communicate with each other.

8.2 Classes and Objects

In Java, a class is a blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of that class will have.

Example: Here's a simple 'Person' class:

```
public class Person {
    // Attributes
    String name;
    int age;

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

    // Method
    public void greet() {
        System.out.println("Hello, my name is " + name);
    }
}
```

In this example, the ‘Person‘ class has attributes (‘name‘ and ‘age‘), a constructor, and a method ‘greet()‘.

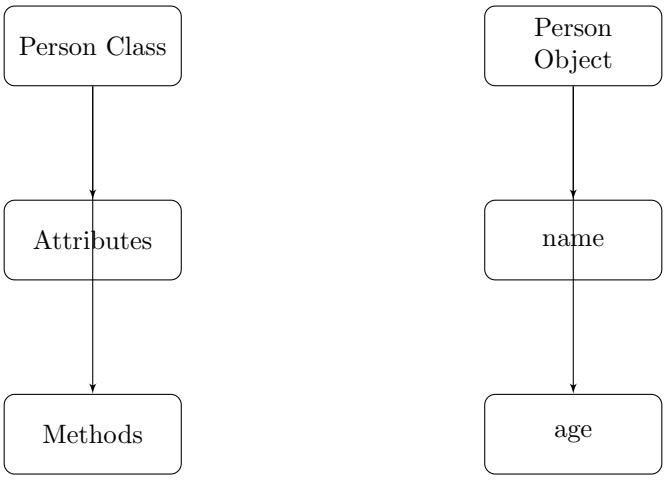

Figure 4: Class and Object Relationship

To create an object of the ‘Person‘ class:

```
Person person1 = new Person("Alice", 30);
```

8.3 Inheritance

Inheritance is a fundamental object-oriented programming concept that allows a class (subclass or child class) to inherit properties and behaviors from another class (superclass or parent class). Inheritance promotes code reusability and the creation of hierarchies of related classes.

Example: Let’s extend the ‘Person‘ class to create a ‘Student‘ class:

```
public class Student extends Person {
    // Additional attributes
    String studentId;

    // Constructor
    public Student(String name, int age, String studentId) {
        super(name, age); // Call the superclass constructor
        this.studentId = studentId;
    }

    // Method
```

```
    public void study() {
        System.out.println(name + " is studying.");
    }
}
```

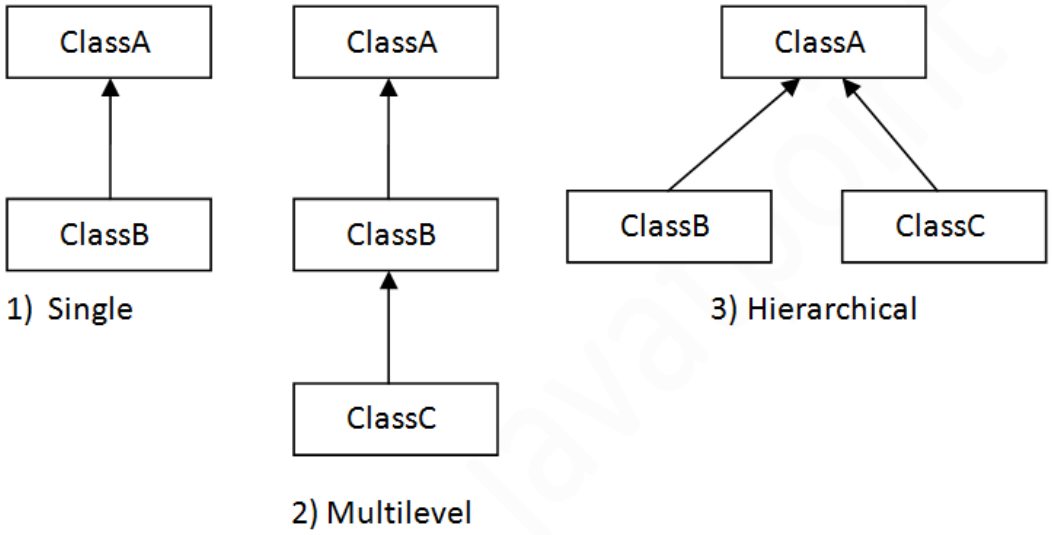

Figure 5: Types of Inheritence

The ‘Student‘ class extends the ‘Person‘ class, inheriting properties such as ‘name‘ and ‘age‘, and also has its own attribute, ‘studentId‘. Additionally, it introduces a method ‘study()‘ that is specific to students.

Inheritance is a powerful mechanism in object-oriented programming, allowing for code organization, reusability, and the creation of specialized classes based on existing ones.

8.4 Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. This promotes flexibility in handling different types of objects.

Example: Using polymorphism with a common interface ‘Shape‘:

```
public interface Shape {
    double calculateArea();
}

public class Circle implements Shape {
    double radius;

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

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

public class Rectangle implements Shape {
    double width;
    double height;

```

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

    @Override
    public double calculateArea() {
        return width * height;
    }
}
```

In this example, both ‘Circle‘ and ‘Rectangle‘ implement the ‘Shape‘ interface, allowing polymorphic behavior.

8.5 Encapsulation

Encapsulation is the concept of bundling data (attributes) and methods (functions) into a single unit called a class. Access to the data is controlled through methods to ensure data hiding and security.

Example: Using encapsulation with access modifiers:

```
public class BankAccount {
    private double balance; // Private attribute

    public BankAccount(double initialBalance) {
        balance = initialBalance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }

    public double getBalance() {
        return balance; // Encapsulated access
    }
}
```

In this example, the ‘balance‘ attribute is private, and access to it is controlled through methods.

8.6 Abstraction

Abstraction is the concept of simplifying complex systems by breaking them into smaller, more manageable parts. It allows you to focus on essential features while hiding non-essential details.

Example: Using abstraction with abstract classes and interfaces:

```
public abstract class Animal {
    public abstract void makeSound();
}

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

```
}

public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow!");
    }
}
```

In this example, the 'Animal' abstract class defines an abstraction for animal sounds, and concrete classes like 'Dog' and 'Cat' provide specific implementations.

8.7 Object-Oriented Programming Benefits

Object-Oriented Programming in Java offers several advantages:
- **Modularity:** Code is organized into classes, making it more manageable and reusable.
- **Reusability:** Classes and objects can be reused in various parts of an application.
- **Maintainability:** Changes to one part of the code do not affect other parts if encapsulation is used correctly.
- **Flexibility:** OOP supports polymorphism, enabling dynamic behavior in programs.
- **Abstraction:** It abstracts complex systems, making them easier to understand and work with.

Understanding and effectively applying OOP principles is crucial for Java developers to create maintainable, scalable, and well-structured code.

9 Sorting Algorithms

Sorting is a fundamental operation in computer science and is used to arrange elements in a specific order. There are various sorting algorithms available, each with its own advantages and disadvantages. In this section, we'll explore some common sorting algorithms and provide example code for one of them.

9.1 Bubble Sort

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the entire list is sorted.

9.1.1 Example Code in Java

Here's an example of the Bubble Sort algorithm implemented in Java:

```
public class BubbleSort {
    public static void main(String[] args) {
        int[] arr = {64, 34, 25, 12, 22, 11, 90};

        // Perform the Bubble Sort
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Swap arr[j] and arr[j+1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }

        // Print the sorted array
        System.out.println("Sorted array:");
        for (int value : arr) {
            System.out.print(value + " ");
```

```
        }
    }
}
```

Bubble Sort is not the most efficient sorting algorithm, but it is easy to understand and implement.

9.2 Optimized Bubble Sort

Optimized Bubble Sort is a variation of the classic Bubble Sort algorithm that includes improvements to reduce the number of comparisons and swaps. It is particularly useful when dealing with partially sorted arrays or lists with a large number of elements already in their correct positions.

9.2.1 Example Code in Java

Here's an example of the Optimized Bubble Sort algorithm implemented in Java:

```
public class OptimizedBubbleSort {
    public static void main(String[] args) {
        int[] arr = {64, 34, 25, 12, 22, 11, 90};

        // Perform Optimized Bubble Sort
        optimizedBubbleSort(arr);

        // Print the sorted array
        System.out.println("Sorted array:");
        for (int value : arr) {
            System.out.print(value + " ");
        }
    }

    static void optimizedBubbleSort(int arr[]) {
        int n = arr.length;
        boolean swapped;

        for (int i = 0; i < n - 1; i++) {
            swapped = false;
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Swap arr[j] and arr[j+1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }
            if (!swapped) {
                // If no two elements were swapped in the inner loop, the array is already sor
                break;
            }
        }
    }
}
```

Optimized Bubble Sort is more efficient than the standard Bubble Sort, especially when dealing with almost sorted data. It has an average-case time complexity of $O(n^2)$, but it can perform better than this in certain situations.

In this example, we provide a Java implementation of the Optimized Bubble Sort algorithm with explanations in the comments.

9.3 Quicksort

Quicksort is a highly efficient sorting algorithm known for its average-case performance. It uses a divide-and-conquer strategy to sort an array by selecting a 'pivot' element and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

9.3.1 Example Code in Java

Here's an example of the Quicksort algorithm implemented in Java:

```
public class QuickSort {
    public static void main(String[] args) {
        int[] arr = {64, 34, 25, 12, 22, 11, 90};

        // Perform the Quicksort
        quickSort(arr, 0, arr.length - 1);

        // Print the sorted array
        System.out.println("Sorted array:");
        for (int value : arr) {
            System.out.print(value + " ");
        }
    }

    static void quickSort(int arr[], int low, int high) {
        if (low < high) {
            int pivotIndex = partition(arr, low, high);

            quickSort(arr, low, pivotIndex - 1);
            quickSort(arr, pivotIndex + 1, high);
        }
    }

    static int partition(int arr[], int low, int high) {
        int pivot = arr[high];
        int i = (low - 1);

        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        return i + 1;
    }
}
```

Quicksort is a widely used sorting algorithm due to its average-case time complexity of O(n log n). It is often preferred when sorting large datasets.

In this example, we provide a Java implementation of the Quicksort algorithm along with comments to explain key steps.

9.4 Insertion Sort

Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time. It is especially efficient for small datasets or nearly sorted data. The algorithm works by repeatedly taking an element from the unsorted part and inserting it into its correct position within the sorted part of the array.

9.4.1 Example Code in Java

Here's an example of the Insertion Sort algorithm implemented in Java:

```
public class InsertionSort {
    public static void main(String[] args) {
        int[] arr = {64, 34, 25, 12, 22, 11, 90};

        // Perform the Insertion Sort
        for (int i = 1; i < arr.length; i++) {
            int key = arr[i];
            int j = i - 1;

            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }

            arr[j + 1] = key;
        }

        // Print the sorted array
        System.out.println("Sorted array:");
        for (int value : arr) {
            System.out.print(value + " ");
        }
    }
}
```

Insertion Sort has an average-case time complexity of $O(n^2)$ and a space complexity of $O(1)$.

In this example, we provide a Java implementation of the Insertion Sort algorithm along with comments to explain the key steps.

9.5 Merge Sort

Merge Sort is a popular and efficient sorting algorithm that uses the divide-and-conquer strategy to sort an array. It divides the unsorted list into n sublists, each containing one element, and repeatedly merges sublists to produce new sorted sublists until there is only one sublist remaining.

9.5.1 Example Code in Java

Here's an example of the Merge Sort algorithm implemented in Java:

```
public class MergeSort {
    public static void main(String[] args) {
        int[] arr = {64, 34, 25, 12, 22, 11, 90};

        // Perform the Merge Sort
        mergeSort(arr, 0, arr.length - 1);

        // Print the sorted array
        System.out.println("Sorted array:");
        for (int value : arr) {
            System.out.print(value + " ");
```

```
        }
    }

    static void mergeSort(int arr[], int l, int r) {
        if (l < r) {
            int m = (l + r) / 2;

            mergeSort(arr, l, m);
            mergeSort(arr, m + 1, r);

            merge(arr, l, m, r);
        }
    }

    static void merge(int arr[], int l, int m, int r) {
        // Merge two subarrays of arr[]
        int n1 = m - l + 1;
        int n2 = r - m;

        int L[] = new int[n1];
        int R[] = new int[n2];

        for (int i = 0; i < n1; i++) {
            L[i] = arr[l + i];
        }
        for (int j = 0; j < n2; j++) {
            R[j] = arr[m + 1 + j];
        }

        // Merge the subarrays
        int i = 0, j = 0, k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        // Copy remaining elements of L[] and R[]
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }
}
```

Merge Sort is known for its stability and average-case time complexity of $O(n \log n)$. It has a space complexity of $O(n)$.

In this example, we provide a Java implementation of the Merge Sort algorithm along with comments to explain the key steps.

10 Stacks

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It means that the most recently added element is the first one to be removed. Stacks are commonly used in programming for managing function calls, tracking execution history, and solving problems that require a last-in, first-out approach.

10.1 Operations on a Stack

A stack typically supports two primary operations:

- **Push**: Adding an element to the top of the stack.
- **Pop**: Removing the top element from the stack.

Additional operations may include:

- **Peek (Top)**: Viewing the top element without removing it.
- **isEmpty**: Checking if the stack is empty.
- **Size**: Finding the number of elements in the stack.

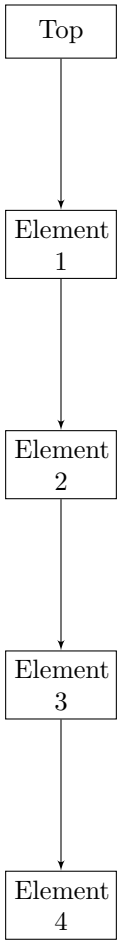

Figure 6: Stack Data Structure (LIFO Order)

10.2 Initiating a Stack in Java

To initiate a stack in Java, you can use the `java.util.Stack` class or a `java.util.LinkedList` to implement a stack. Here's an example of how to create a stack using the `java.util.Stack` class:

```
import java.util.Stack;

public class StackExample {
    public static void main(String[] args) {
        // Create a stack
        Stack<Integer> stack = new Stack<>();

        // Push elements onto the stack
        stack.push(10);
        stack.push(20);
        stack.push(30);

        // Pop elements from the stack
        int poppedElement = stack.pop();
        System.out.println("Popped element: " + poppedElement);

        // Peek at the top element
        int topElement = stack.peek();
        System.out.println("Top element: " + topElement);

        // Check if the stack is empty
        boolean isEmpty = stack.isEmpty();
        System.out.println("Is the stack empty? " + isEmpty);
    }
}
```

11 Queues

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. It means that the first element added is the first one to be removed. Queues are commonly used in programming for managing tasks, scheduling, and solving problems that require a first-in, first-out approach.

11.1 Operations on a Queue

A queue typically supports two primary operations:

- **Enqueue**: Adding an element to the rear (end) of the queue.
- **Dequeue**: Removing an element from the front of the queue.

Additional operations may include:

- **Peek (Front)**: Viewing the front element without removing it.
- **isEmpty**: Checking if the queue is empty.
- **Size**: Finding the number of elements in the queue.

11.2 Diagram of a Queue

The diagram above illustrates a queue data structure with labeled elements, a "Front" pointer, and a "Rear" pointer. Elements are enqueued at the rear and dequeued from the front, following the First-In-First-Out (FIFO) order.

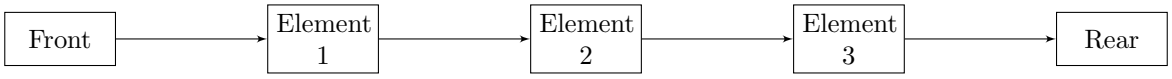

Figure 7: Queue Data Structure (FIFO Order)

11.3 Initiating a Queue in Java

To initiate a queue in Java, you can use the 'java.util.LinkedList' class, or you can use a specific data structure like 'java.util.Queue'. Here's an example of how to create a queue using the 'java.util.LinkedList' class:

```
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args) {
        // Create a queue
        Queue<Integer> queue = new LinkedList<>();

        // Enqueue elements into the queue
        queue.add(10);
        queue.add(20);
        queue.add(30);

        // Dequeue elements from the queue
        int dequeuedElement = queue.remove();
        System.out.println("Dequeued element: " + dequeuedElement);

        // Peek at the front element
        int frontElement = queue.peek();
        System.out.println("Front element: " + frontElement);

        // Check if the queue is empty
        boolean isEmpty = queue.isEmpty();
        System.out.println("Is the queue empty? " + isEmpty);
    }
}
```

12 Linked Lists

A linked list is a linear data structure consisting of a sequence of elements where each element points to the next one. Linked lists provide dynamic memory allocation, efficient insertions, and deletions, making them a fundamental data structure in computer science.

12.1 Types of Linked Lists

There are several types of linked lists:

- **Singly Linked List**: Each element (node) points to the next element in the list. The last element points to 'null'.
- **Doubly Linked List**: Each element has pointers to both the next and the previous elements. It allows traversal in both directions.
- **Circular Linked List**: In this list, the last element points to the first element, forming a closed loop.

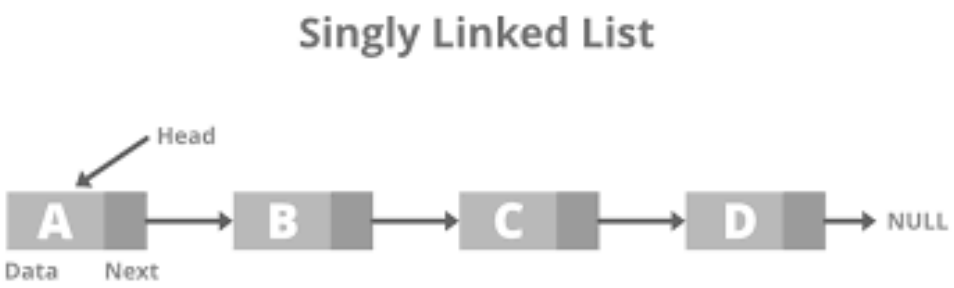

Figure 8: Singly LinkedList

12.2 Diagram of a Singly Linked List

12.3 Common Operations on Linked Lists

Linked lists support various operations, including:

- **Insertion**: Adding a new element to the list.
- **Deletion**: Removing an element from the list.
- **Traversal**: Iterating through the list.
- **Searching**: Finding an element in the list.

Linked lists are versatile data structures used in various applications, such as implementing dynamic data structures, managing memory, and building more complex data structures like stacks and queues.

12.4 Initiating a Linked List in Java

To initiate a linked list in Java, you can use the 'java.util.LinkedList' class or implement your own linked list. Here's an example of how to create a linked list using the 'java.util.LinkedList' class:

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        // Create a linked list
        LinkedList<String> linkedList = new LinkedList<>();

        // Add elements to the linked list
        linkedList.add("Apple");
        linkedList.add("Banana");
        linkedList.add("Cherry");

        // Access elements in the linked list
        String firstElement = linkedList.getFirst();
        String lastElement = linkedList.getLast();

        // Print elements
        System.out.println("First Element: " + firstElement);
        System.out.println("Last Element: " + lastElement);

        // Check if the linked list is empty
        boolean isEmpty = linkedList.isEmpty();
        System.out.println("Is the linked list empty? " + isEmpty);
    }
}
```

13 Practice

In this section, we provide a set of practice questions at different difficulty levels to test your Java programming skills. Whether you're a beginner, intermediate, or advanced programmer, you can find questions that suit your level. Try to solve these exercises to reinforce your understanding of Java.

13.1 Beginner

1. Write a Java program to print "Hello, World!" to the console.
2. Calculate and print the sum of two integers entered by the user.
3. Create a program to find and print the largest number among three integers.
4. Write a Java program to check if a given number is even or odd.
5. Develop a program to calculate the factorial of a positive integer.
6. Create a simple calculator program that can perform addition, subtraction, multiplication, and division.
7. Write a Java program to find the Fibonacci series for a specified number of terms.
8. Implement a program to reverse a string entered by the user.
9. Create a program that counts the number of vowels in a given string.
10. Develop a program to print the prime numbers within a specified range.

13.2 Intermediate

1. Implement a stack data structure in Java with operations for push, pop, and checking if it's empty.
2. Create a program to check if a given string is a palindrome (reads the same forwards and backward).
3. Write a Java program to find the factorial of a number using both iterative and recursive methods.
4. Implement a linked list data structure in Java with operations for insertion, deletion, and traversing the list.
5. Develop a Java program to perform binary search in a sorted array.
6. Create a program to find the common elements between two arrays.
7. Implement a basic sorting algorithm (e.g., bubble sort or selection sort) in Java.
8. Write a program to reverse a linked list.
9. Design and implement a simple binary tree with methods for insertion and traversal.
10. Develop a Java program to solve the Tower of Hanoi problem with 'n' disks.

13.3 Advanced

1. Implement a hash table data structure in Java with methods for insertion, retrieval, and collision handling.
2. Create a program to perform depth-first search (DFS) and breadth-first search (BFS) on a graph.
3. Write a Java program to find the shortest path in a weighted graph using Dijkstra's algorithm.
4. Implement a binary search tree (BST) with methods for insertion, deletion, and finding the k-th smallest element.
5. Develop a program to perform quicksort or mergesort on an array.
6. Design and implement a red-black tree in Java.
7. Create a program to find the longest common subsequence of two strings.

8. Implement a graph data structure and write an algorithm to find the strongly connected components in a directed graph.
9. Write a program to solve the traveling salesman problem (TSP) using a suitable algorithm.
10. Develop a Java program to perform topological sorting on a directed acyclic graph (DAG).