q:-

1.1.1 Introduction to programming

paradigm, types of programming

paradigm.

ans:-

Introduction to Programming Paradigms:

Programming paradigm refers to the approach or style of programming that guides the structure, design, and implementation of computer programs. Each programming paradigm has its own set of principles, concepts, and techniques for solving problems. In this essay, we will discuss three commonly used programming paradigms: procedural, object-oriented, and functional.

1. Procedural Programming Paradigm:

Procedural programming focuses on the step-by-step execution of a sequence of instructions. It uses procedures or functions to perform specific tasks. An example of procedural programming in Java is shown below:

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

```
    public static int addNumbers(int x, int y) {
        return x + y;
    }
}
```

Output:

```
Sum: 8
```

In the above example, we define a `main` function that calls the `addNumbers` function to add two numbers and stores the result in the `sum` variable. The result is then printed to the console.

2. Object-Oriented Programming Paradigm:

Object-oriented programming (OOP) focuses on creating objects that encapsulate data and behavior. It promotes the concept of classes and objects, allowing for modular and reusable code. Here's an example of object-oriented programming in Java:

```java
public class Rectangle {
    private int length;
    private int width;

    public Rectangle(int length, int width) {
        this.length = length;
```

```
        this.width = width;
    }

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

public class OOPExample {
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle(5, 3);
        int area = rectangle.calculateArea();
        System.out.println("Area: " + area);
    }
}
```

Output:

```
Area: 15
```

In the above example, we define a `Rectangle` class with `length` and `width` attributes. The class has a constructor to initialize these attributes and a `calculateArea` method to compute the area of the rectangle. In the `main` function, we create an instance of the `Rectangle` class and call the `calculateArea` method to get the area, which is then printed.

3. Functional Programming Paradigm:

Functional programming focuses on writing programs using pure functions that avoid changing state or modifying data. It emphasizes immutability and the use of higher-order functions. Here's an example of functional programming in Java using lambda expressions:

```java
import java.util.Arrays;
import java.util.List;

public class FunctionalExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        int sum = numbers.stream()
                .filter(n -> n % 2 == 0)
                .mapToInt(n -> n)
                .sum();
        System.out.println("Sum of even numbers: " + sum);
    }
}
```

Output:

```
Sum of even numbers: 6
```

In the above example, we create a list of numbers and use the Java Stream API to perform functional operations. We filter the even numbers, map them to integers, and calculate the sum using the `sum` method. The result is then printed to the console.

These examples demonstrate three different programming paradigms: procedural, object-oriented, and functional. Each paradigm offers its own benefits and is suitable for different types of problems. Understanding these paradigms allows programmers to choose the most appropriate approach when developing software.

q:-

1.2.1 Procedure-Oriented vs. Object-

Oriented Programming concept

a:-

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP,Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function,it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

q:-

1.3.1 Basics of OOP: Class, Object, data

abstraction, encapsulation,

inheritance, benefits of

inheritance, polymorphism.

a:-

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of objects, which are instances of classes. OOP offers several fundamental concepts, including classes, objects, data abstraction, encapsulation, inheritance, benefits of inheritance, and polymorphism. Let's explore each of these concepts with suitable examples in Java:

1. Class: A class is a blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that an object of that class can exhibit. For example:

```java
class Car {
    String brand;
    String color;

    void startEngine() {
        System.out.println("Engine started!");
    }
}
```

2. Object: An object is an instance of a class. It represents a specific entity that has its own state and behavior. For example:

```java
Car myCar = new Car();
myCar.brand = "Toyota";
myCar.color = "Red";
myCar.startEngine();
```

Output: "Engine started!"

3. Data Abstraction: Data abstraction refers to the process of hiding the internal details of an object and only exposing the essential features to the outside world. It helps in providing a simplified interface for the user. For example:

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

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

```
Shape shape = new Circle();

shape.draw();
```

Output: "Drawing a circle"

4. Encapsulation: Encapsulation is the mechanism of bundling data and methods together within a class, such that the data is hidden from external access and can only be accessed through defined methods. It provides data security and maintains code maintainability. For example:

```java
class BankAccount {
    private double balance;

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

    public double getBalance() {
        return balance;
    }
}

BankAccount account = new BankAccount();

account.deposit(1000);

System.out.println("Balance: " + account.getBalance());
```

Output: "Balance: 1000.0"

5. Inheritance: Inheritance allows the creation of a new class (derived class) that inherits properties and methods from an existing class (base class). It promotes code reuse and supports the concept of a hierarchy. For example:

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

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

Dog myDog = new Dog();
myDog.eat();
myDog.bark();
```

Output:

"Animal is eating"

"Dog is barking"

6. Benefits of Inheritance: Inheritance provides code reuse, improves code organization, and allows for the creation of specialized classes based on existing ones. It also supports the concept of polymorphism, which we'll discuss next.

7. Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It provides the ability to perform a single action in different ways based on the object type. For example:

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

class Cat extends Animal {
    void makeSound() {
        System.out.println("Cat meows");
    }
}

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

```
Animal myAnimal1 = new Cat();

Animal myAnimal2 = new Dog();


myAnimal1.makeSound();

myAnimal2.makeSound();
```

Output:

"Cat meows"

"Dog barks"

These are the basics of OOP, including class, object, data abstraction, encapsulation, inheritance, the benefits of inheritance, and polymorphism. Understanding these concepts is crucial for building modular and extensible software systems.

q:-

1.4.1 Basics of Java, Background/History

of Java, Java and the Internet,

Advantages of Java.

a:-

Java is a popular programming language that was created by James Gosling and his team at Sun Microsystems in the mid-1990s. It was initially developed to address the limitations of other programming languages and to provide a platform-independent solution. Here are the basics of Java, its background/history, its relationship with the internet, and its advantages:

1. Basics of Java (1 mark):

- Java is an object-oriented programming language that follows the "write once, run anywhere" principle.

- It uses a virtual machine called the Java Virtual Machine (JVM) to execute Java bytecode.

2. Background/History of Java (2 marks):

- Java was introduced in 1995 and quickly gained popularity due to its simplicity and versatility.

- It was originally designed for consumer electronics, but its potential was recognized for internet programming.

3. Java and the Internet (2 marks):

- Java was instrumental in the development of internet-based applications and applets.

- Applets are small Java programs that can be embedded in webpages and executed within a web browser.

4. Advantages of Java (2 marks):

- Platform Independence: Java programs can run on any platform that has a JVM, providing flexibility and portability. For example, a Java program written on Windows can run on Linux or macOS without any modifications.

- Object-Oriented: Java's object-oriented nature allows for modular and reusable code, enhancing productivity and maintainability. For example, you can create a class in Java and reuse it in multiple projects.

- Memory Management: Java uses automatic memory management through garbage collection, which helps developers focus on writing code rather than managing memory explicitly. For example, Java automatically deallocates memory for objects that are no longer in use.

Example:

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

```
  }
}
```

Output:

```
Hello, World!
```

In the example above, we have a simple Java program that prints "Hello, World!" to the console. This demonstrates the basic structure of a Java program with a class, a main method, and a print statement. When the program is executed, the output is displayed on the console.

q:-

1.4.2 JDK, JRE, JVM, and Byte code.

a:-

JDK, JRE, JVM, and bytecode are essential components of the Java programming language. Each one plays a specific role in the development and execution of Java applications. Let's explore each of them in detail:

1. JDK (Java Development Kit):

The JDK is a software development kit provided by Oracle, which includes tools necessary for developing Java applications. It consists of the Java compiler, debugger, libraries, and other utilities required for Java programming. The JDK provides a development environment for writing, compiling, and testing Java code. For example, the JDK provides the "javac" command-line tool to compile Java source code into bytecode.

2. JRE (Java Runtime Environment):

The JRE is a runtime environment that provides the necessary libraries and components to run Java applications. It includes the JVM (Java Virtual Machine) and the core Java class libraries. The JRE is required to execute Java applications on a target machine. For instance, if you want to run a Java program on your computer, you need to have the JRE installed. It does not include development tools like compilers or debuggers.

3. JVM (Java Virtual Machine):

The JVM is the cornerstone of the Java platform. It is an abstract machine that executes Java bytecode. It acts as an interpreter between the compiled Java bytecode and the host operating system. The JVM provides a runtime environment for executing Java applications. It performs tasks like memory management, garbage collection, and bytecode interpretation. For example, the JVM translates Java bytecode into machine-specific instructions for the operating system to execute.

4. Bytecode:

Bytecode is the intermediate representation of Java source code. When you compile a Java program, the Java compiler converts the source code into bytecode. Bytecode is a low-level format that is platform-independent, meaning it can be executed on any machine with a compatible JVM. For example, consider the following Java code:

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

After compilation, it generates bytecode that looks something like this:

```

0:  getstatic     #2

3:  ldc           #3

5:  invokevirtual #4

8:  return

```

The bytecode can then be executed by the JVM, which interprets and runs the instructions, resulting in the output: "Hello, World!".

In conclusion, the JDK provides development tools, the JRE allows you to run Java applications, the JVM executes bytecode, and bytecode is the compiled form of Java source code. Together, these components enable the development and execution of Java programs on various platforms.

q:-

1.4.4 Java program structure.

a:-

Java Program Structure:

1. Package Declaration: The package declaration specifies the package in which the Java class is located. It is the first line of code in a Java file and is optional. It helps in organizing and categorizing classes into different packages.

Example:

```java

package com.example.myapp;

```

2. Import Statements: Import statements are used to include classes or packages from other packages into the current Java file. They enable the usage of classes without specifying their fully qualified names.

Example:

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

3. Interface Section: Interfaces define a contract that classes can implement. It contains abstract methods that must be implemented by the classes implementing the interface.

Example:

```java
public interface MyInterface {
    void myMethod();
}
```

4. Class Definition: A class is a blueprint or template that defines the properties (variables) and behaviors (methods) of objects. It is the fundamental building block in Java programming.

Example:

```java
public class MyClass {
    // Class members (variables and methods) go here
```

}

```

5. Class Variables and Variables: Class variables (also known as static variables) are shared among all instances of a class, while variables (instance variables) are unique to each instance of the class.

Example:

```java
public class MyClass {
    // Class variable
    public static int classVariable = 10;

    // Instance variable
    private int instanceVariable;
}
```

6. Main Method Class: The main method is the entry point for a Java program. It is where the program starts executing. It must have a specific signature: `public static void main(String[] args)`.

Example:

```java
public class Main {
    public static void main(String[] args) {
        // Code to be executed
```
```

```
    }
}
```

7. Methods and Behaviors: Methods are blocks of code that perform specific tasks. They encapsulate behavior and can be called to perform a particular action.

Example:

```java
public class MyClass {
    public void myMethod() {
        // Code for the method
    }
}
```

Overall, a complete Java program structure includes package declaration, import statements, interface section (optional), class definition, class variables and variables, the main method class, and other methods and behaviors. Each element plays a crucial role in defining the structure and functionality of a Java program.

Note: The output of a Java program depends on the specific code written in the program. It may involve printing messages, performing calculations, or interacting with the user, among other things. The output cannot be generalized without specific code.

q:-

1.4.5 Compiling and running a simple java program.

a:-

Compiling and running a simple Java program typically involves several steps. Below, I'll outline the process in seven steps, providing examples and expected outputs along the way. Please note that due to the space limitations of 20 lines, the code examples will be condensed.

Step 1: Write the Java program

Create a Java file with a .java extension, such as "MyProgram.java," and open it in a text editor. Write a simple program, like the one shown below:

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

Step 2: Save the file

Save the file with a .java extension in a directory of your choice. For example, save it as "MyProgram.java."

Step 3: Open a terminal or command prompt

Open a terminal or command prompt on your computer.

Step 4: Navigate to the directory

Use the "cd" command to navigate to the directory where you saved the Java file. For example, if you saved it on the desktop, you would run the following command:

```

cd Desktop

```

Step 5: Compile the Java file

In the terminal or command prompt, compile the Java file using the "javac" command followed by the filename. For example:

```

javac MyProgram.java

```

If the compilation is successful, you won't see any output. Otherwise, error messages will be displayed, indicating any issues in the code.

Step 6: Run the Java program

After the Java file is compiled successfully, run the program using the "java" command followed by the class name (without the .java extension). For example:

```

java MyProgram

```

The program will execute, and you should see the output in the terminal or command prompt:

```
```

Hello, World!

```

Step 7: Analyze the output

The output of the program should match the expected result, which in this case is "Hello, World!"

Remember, these steps provide a basic example of compiling and running a Java program. In more complex scenarios involving external libraries or multiple classes, additional steps may be required.

q:-

1.4.6 Comments in java, data types,

variables, scope and lifetime of

variables, operators, type

conversion and casting, Control

Statements – If, else, nested if, if-

else ladders, Switch, while, do-

while, for, for-each, break,

continue, arrays in java.
```

a:-

Comments in Java:

In Java, comments are used to add explanatory text within the code that is ignored by the compiler. They help in improving code readability and understanding. There are two types of comments in Java: single-line comments and multi-line comments.

Single-line comments start with // and are used to comment a single line of code. For example:

```java
int age = 25; // This variable stores the age of a person
```

Multi-line comments start with /* and end with */ and can span across multiple lines. They are used to comment a block of code. For example:

```java
/*
This method calculates the sum of two numbers.
@param a the first number
@param b the second number
@return the sum of a and b
*/
public int sum(int a, int b) {
    return a + b;
}
```

Data Types:

Java has two categories of data types: primitive data types and reference data types.

Primitive data types represent basic values and include:

- Integer types: byte, short, int, long

- Floating-point types: float, double

- Character type: char

- Boolean type: boolean

For example:

```java

int age = 25;

double salary = 5000.50;

char grade = 'A';

boolean isActive = true;

```

Reference data types are derived from classes and include:

- Class types: String, Scanner, etc.

- Array types

Variables:

Variables are used to store data in a program. They have a data type, a name, and a value. The name of a variable should be meaningful and follow naming conventions.

For example:

```java

int age = 25;
```

String name = "John";

```

Scope and Lifetime of Variables:

The scope of a variable determines its visibility and accessibility within the program. Java has three scopes: local, instance, and class.

Local variables are declared inside a method or block and have the narrowest scope. They are accessible only within that method or block. Once the method or block ends, the local variable is destroyed.

For example:

```java
public void printNumber() {
    int number = 10; // local variable
    System.out.println(number);
}
```

Output: 10

Instance variables are declared within a class but outside any method. They belong to the instance of the class and can be accessed by any method within the class.

For example:

```java
public class Person {
    String name; // instance variable
```
```

}

```

Class variables are declared with the keyword "static" and belong to the class itself rather than any specific instance of the class. They are shared by all instances of the class.

For example:

```java
public class Counter {
    static int count; // class variable
}
```

Operators:

Java supports various operators for performing different operations, such as arithmetic, assignment, comparison, logical, etc.

Arithmetic operators perform mathematical calculations:

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Modulus: %

For example:

```java

```
int a = 10;

int b = 5;

int sum = a + b;

int product = a * b;

int modulus = a % b;
```

Type Conversion and Casting:

Type conversion refers to converting one data type to another. Java supports two types of type conversions: implicit (automatic) and explicit (casting).

Implicit type conversion occurs when a smaller data type is assigned to a larger data type without loss of information. For example, assigning an int to a double.

Explicit type conversion or casting is required when a larger data type is assigned to a smaller data type or when converting between incompatible types. Casting is done using parentheses and the desired data type.

For example:

```java
double d = 3.14;

int i = (int) d; // Explicit casting
```

Note: Casting from a larger data type to a smaller data type may result in data loss or truncation.

These are the main concepts related to comments, data types, variables, scope and lifetime of

variables, operators, and type conversion in Java.

Control Statements in Java:

1. If Statement:

The if statement allows you to execute a block of code conditionally based on a given condition. It has the following syntax:

```
if (condition) {
    // code to be executed if the condition is true
}
```

Example:

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

Output: "Number is positive."

2. Else Statement:

The else statement is used with the if statement to specify the code to be executed when the if condition is false. It has the following syntax:

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

Example:

```java
int num = -5;
if (num > 0) {
    System.out.println("Number is positive.");
} else {
    System.out.println("Number is non-positive.");
}
```

Output: "Number is non-positive."

3. Nested If Statement:

A nested if statement is an if statement inside another if statement. It allows you to check for multiple conditions. It has the following syntax:

```
if (condition1) {
```

```
    // code to be executed if condition1 is true
    if (condition2) {
        // code to be executed if both condition1 and condition2 are true
    }
}
```

Example:

```java
int num = 10;
if (num > 0) {
    System.out.println("Number is positive.");
    if (num % 2 == 0) {
        System.out.println("Number is even.");
    }
}
```

Output:

Number is positive.

Number is even.

4. If-else ladder:

An if-else ladder is a series of if-else statements. It allows you to check multiple conditions and execute different code blocks based on those conditions. It has the following syntax:

```
```

```
if (condition1) {
    // code to be executed if condition1 is true
} else if (condition2) {
    // code to be executed if condition1 is false and condition2 is true
} else {
    // code to be executed if all conditions are false
}
```

Example:

```java
int num = 0;
if (num > 0) {
    System.out.println("Number is positive.");
} else if (num < 0) {
    System.out.println("Number is negative.");
} else {
    System.out.println("Number is zero.");
}
```

Output: "Number is zero."

5. Switch Statement:

The switch statement allows you to execute different code blocks based on different possible values of a variable. It has the following syntax:

```
switch (variable) {
    case value1:
        // code to be executed if variable equals value1
        break;
    case value2:
        // code to be executed if variable equals value2
        break;
    default:
        // code to be executed if none of the cases match
}
```

Example:

```java
int day = 3;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
```

```
        break;
    default:
        System.out.println("Invalid day");
}
```

Output: "Wednesday"

6. While Loop:

The while loop allows you to repeatedly execute a block of code as long as a given condition is true. It has the following syntax:

```
while (condition) {
    // code to be executed while the condition is true
}
```

Example:

```java
int count = 1;
while (count <= 5) {
    System.out.println("Count: " + count);
    count++;
}
```

Output:

Count: 1

Count: 2

Count: 3

Count: 4

Count: 5

7. For Loop:

The for loop allows you to execute a block of code repeatedly with a specified number of iterations. It has the following syntax:

```
for (initialization; condition; update) {
    // code to be executed in each iteration
}
```

Example:

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

Output:

Count: 1

Count: 2

Count: 3

Count: 4

Count: 5

These control statements provide powerful mechanisms to control the flow of execution in Java programs, making them more flexible and efficient.

1. for-each loop in Java:

The for-each loop in Java provides an easy way to iterate over elements in an array or collection. It is also known as the enhanced for loop. Here's an example:

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

for (int num : numbers) {
    System.out.println(num);
}
```

Output:

```
1
2
3
4
5
```

In the above example, the for-each loop iterates over each element in the `numbers` array and assigns it to the variable `num`. The loop body then prints the value of `num`. This loop

automatically handles the iteration and eliminates the need for an index variable.

2. break statement in Java:

The `break` statement in Java is used to terminate the execution of a loop or switch statement. When encountered, it immediately exits the innermost loop or switch statement. Here's an example:

```java
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break;
    }
    System.out.println(i);
}
```

Output:

```
1
2
3
4
```

In the above example, the `break` statement is used to exit the `for` loop when the value of `i` is equal to 5. As a result, the loop terminates prematurely, and only the numbers 1 to 4 are printed.

3. continue statement in Java:

The `continue` statement in Java is used to skip the current iteration of a loop and proceed to the next iteration. When encountered, it jumps to the next iteration without executing the remaining statements in the loop body. Here's an example:

```java
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue;
    }
    System.out.println(i);
}
```

Output:

```
1
2
4
5
```

In the above example, the `continue` statement is used to skip the iteration when `i` is equal to 3. As a result, the number 3 is not printed, and the loop proceeds to the next iteration.

4. Arrays in Java:

Arrays in Java are used to store multiple values of the same data type in a single variable. They

provide a way to group related elements and access them using an index. Here's an example of declaring and initializing an array:

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

In the above example, an array named `numbers` is declared and initialized with five elements. The index of the elements starts from 0, so `numbers[0]` refers to the first element, which is 1, and `numbers[4]` refers to the last element, which is 5.

Overall, these concepts are fundamental to programming in Java and are essential for efficient looping and working with collections of data.

1.4.7 Command Line Argument:

In Java, command line arguments are the inputs provided to a Java program when it is run from the command line. These arguments are passed as strings and can be accessed by the main method of the program. Command line arguments allow users to customize the behavior of a program without modifying its source code.

Example:

Let's consider a simple Java program that takes two integers as command line arguments and calculates their sum:

```java
public class CommandLineArgsExample {
    public static void main(String[] args) {
        if (args.length >= 2) {
```

```
            int num1 = Integer.parseInt(args[0]);
            int num2 = Integer.parseInt(args[1]);
            int sum = num1 + num2;
            System.out.println("The sum of " + num1 + " and " + num2 + " is: " + sum);
        } else {
            System.out.println("Please provide two integers as command line arguments.");
        }
    }
}
```

Output:

If we run the above program from the command line with two integers as arguments, we would get the following output:

```
$ java CommandLineArgsExample 10 20
The sum of 10 and 20 is: 30
```

Explanation:

In the above example, the `main` method accepts a `String` array `args` as a parameter. This array contains the command line arguments passed to the program. We check if the length of `args` is at least 2 (indicating that two integers have been provided). If it is, we parse the first and second elements of `args` as integers using `Integer.parseInt()`. Then, we calculate the sum and display it as output. If fewer than two arguments are provided, an error message is displayed.

1.4.8 Garbage Collection:

Garbage collection is an automatic memory management process in Java where the JVM (Java Virtual Machine) reclaims memory occupied by objects that are no longer in use. It frees up memory by identifying and removing objects that are unreachable and no longer referenced by the program.

Types of Garbage Collection:

1. **Serial Garbage Collector**: It uses a single thread for garbage collection and is suitable for small-scale applications. It pauses the application's execution during garbage collection.

2. **Parallel Garbage Collector**: It uses multiple threads for garbage collection, providing improved performance by utilizing multiple processors. It also pauses the application during garbage collection.

3. **Concurrent Mark and Sweep (CMS) Garbage Collector**: It aims to minimize the pause time during garbage collection by executing most of the garbage collection work concurrently with the application threads.

4. **Garbage First (G1) Garbage Collector**: It is designed to provide consistent garbage collection pauses while achieving high throughput. It divides the heap into multiple regions and collects the regions with the most garbage first.

Example:

Let's consider a scenario where a Java program creates multiple objects but does not explicitly deallocate memory for them. The garbage collector automatically identifies and frees the memory occupied by these objects.

```java
public class GarbageCollectionExample {
    public static void main(String[] args) {
        for (int i = 0; i < 100000; i++) {
            new GarbageObject();
```

```
        }
        System.out.println("Memory-consuming objects created.");

        System.gc(); // Request garbage collection

        System.out.println("Garbage collection completed.");
    }
}

class GarbageObject {
    // Some instance variables and methods
    // ...

    @Override
    protected void finalize() throws Throwable {
        // Finalize method called before the object is garbage collected
        System.out.println("Garbage collection for this object.");
        super.finalize();
    }
}
```

Output:

When the above program is executed, the output might be as follows:

```

Memory-consuming objects created.

Garbage collection for this object.

Garbage collection for this object.

...

Garbage collection completed.

```

Explanation:

In the above example, the `GarbageCollectionExample` program creates multiple `GarbageObject` instances inside a loop. Since the `GarbageObject` class overrides the `finalize()` method, it gets called before the object is garbage collected.

After creating the objects, we explicitly call `System.gc()` to request garbage collection. This is not necessary as the JVM automatically triggers garbage collection when it deems appropriate. However, calling `System.gc()` is a way to suggest that garbage collection should be performed at that point.

The program output shows that the `finalize()` method of the `GarbageObject` class is called for each object that is garbage collected.

Note: The exact behavior of garbage collection can vary based on the JVM implementation and runtime environment. The example above demonstrates the concept but may not produce the exact output on all systems.

---

ch-5

5.1.1 Introduction to Stream, types of

Stream.

Introduction to Stream:

A stream is a sequence of data elements that can be processed sequentially. It is a fundamental concept in programming and is commonly used for input/output (I/O) operations. Streams provide an abstraction layer that allows programmers to interact with various sources or destinations of data, such as files, network connections, or memory buffers, in a uniform way.

In Java, streams are used to efficiently handle input and output operations. There are two main types of streams: byte streams and character streams.

1. Byte Streams:

Byte streams are used for input and output of raw bytes. They are represented by classes that end with `InputStream` or `OutputStream`. The byte stream classes provide methods for reading and writing bytes. Here are two examples of byte stream classes:

a) `FileInputStream` - This class is used to read bytes from a file. It provides methods such as `read()` to read a single byte and `read(byte[] buffer)` to read multiple bytes into a buffer. Here's an example of reading bytes from a file:

```java
try (FileInputStream fis = new FileInputStream("example.txt")) {
```

```
    int byteData;
    while ((byteData = fis.read()) != -1) {
        // Process the byte data
        System.out.print((char) byteData);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Output:

```
This is an example file.
```

b) `FileOutputStream` - This class is used to write bytes to a file. It provides methods such as `write(int b)` to write a single byte and `write(byte[] buffer)` to write multiple bytes from a buffer. Here's an example of writing bytes to a file:

```java
try (FileOutputStream fos = new FileOutputStream("output.txt")) {
    String data = "This is some data.";
    byte[] byteData = data.getBytes();
    fos.write(byteData);
} catch (IOException e) {
    e.printStackTrace();
```

}

```

This code writes the string "This is some data." as bytes to a file named "output.txt".

2. Character Streams:

Character streams are used for input and output of characters. They are represented by classes that end with `Reader` or `Writer`. The character stream classes provide methods for reading and writing characters. Here are two examples of character stream classes:

a) `FileReader` - This class is used to read characters from a file. It provides methods such as `read()` to read a single character and `read(char[] buffer)` to read multiple characters into a buffer. Here's an example of reading characters from a file:

```java
try (FileReader reader = new FileReader("example.txt")) {
    int charData;
    while ((charData = reader.read()) != -1) {
        // Process the character data
        System.out.print((char) charData);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Output:
```

```

This is an example file.

```

b) `FileWriter` - This class is used to write characters to a file. It provides methods such as `write(int c)` to write a single character and `write(char[] buffer)` to write multiple characters from a buffer. Here's an example of writing characters to a file:

```java
try (FileWriter writer = new FileWriter("output.txt")) {
    String data = "This is some data.";
    writer.write(data);
} catch (IOException e) {
    e.printStackTrace();
}
```

This code writes the string "This is some data." as characters to a file named "output.txt".

These examples illustrate how byte streams and character streams can be used to perform input and output operations in Java.

5.1.2 Stream classes and its hierarchy,

I/O classes: File Class, File

InputStream, File Output Stream,

InputStreamReader,

OutputStreamWriter, FileReader,

FileWriter, Buffered Reader.

Stream classes in Java are used for performing input/output operations with various data sources, such as files, network connections, and memory. They provide a convenient way to handle input and output operations efficiently. Here are the definitions and four methods of each of the mentioned stream classes:

1. File Class:

   Definition: The File class represents a file or directory on the file system.

   Methods:

   a) exists(): Checks whether the file or directory exists.

   b) isFile(): Determines if the path represents a file.

   c) isDirectory(): Determines if the path represents a directory.

   d) delete(): Deletes the file or directory.

   Example:

   ```java
   File file = new File("example.txt");
   System.out.println(file.exists());      // Output: false
   System.out.println(file.isFile());      // Output: false
   System.out.println(file.isDirectory()); // Output: false
   ```

2. FileInputStream:

Definition: The FileInputStream class is used to read data from a file as a sequence of bytes.

Methods:

a) read(): Reads a byte of data from the input stream.

b) available(): Returns an estimate of the number of remaining bytes that can be read from the input stream.

c) skip(long n): Skips over and discards n bytes of data from the input stream.

d) close(): Closes the input stream.

Example:

```java
FileInputStream fis = new FileInputStream("example.txt");
int data = fis.read();
System.out.println(data); // Output: 104 (ASCII value of 'h')
fis.close();
```

3. FileOutputStream:

Definition: The FileOutputStream class is used to write data to a file as a sequence of bytes.

Methods:

a) write(int b): Writes a byte to the output stream.

b) flush(): Flushes the output stream, ensuring that any buffered data is written to the file.

c) close(): Closes the output stream.

Example:

```java
FileOutputStream fos = new FileOutputStream("example.txt");
fos.write(104); // Writes the ASCII value of 'h' to the file
fos.close();
```

4. InputStreamReader:

Definition: The InputStreamReader class is a bridge from byte streams to character streams. It reads bytes from an InputStream and decodes them into characters using a specified charset.

Methods:

a) read(): Reads a single character from the input stream.

b) ready(): Checks whether the input stream is ready to be read.

c) skip(long n): Skips over and discards n characters from the input stream.

d) close(): Closes the input stream.

Example:

```java
FileInputStream fis = new FileInputStream("example.txt");
InputStreamReader isr = new InputStreamReader(fis);
int data = isr.read();
System.out.println((char) data); // Output: 'h'
isr.close();
```

5. OutputStreamWriter:

Definition: The OutputStreamWriter class is a bridge from character streams to byte streams.

It writes characters to an OutputStream and encodes them into bytes using a specified charset.

Methods:

a) write(int c): Writes a single character to the output stream.

b) flush(): Flushes the output stream, ensuring that any buffered characters are written to the file.

c) close(): Closes the output stream.

Example:

```java
FileOutputStream fos = new FileOutputStream("example.txt");
OutputStreamWriter osw = new OutputStreamWriter(fos);
osw.write('h'); // Writes the character 'h' to the file
osw.close();
```

6. FileReader:

Definition: The FileReader class is used to read data from a file as a sequence of characters.

Methods:

a) read(): Reads a single character from the input stream.

b) close(): Closes the input stream.

Example:

```java
FileReader fr = new FileReader("example.txt");
int data = fr.read();
System.out.println((char) data); // Output: 'h'
```

```
fr.close();
```

7. FileWriter:

Definition: The FileWriter class is used to write data to a file as a sequence of characters.

Methods:

a) write(String str): Writes a string of characters to the output stream.

b) flush(): Flushes the output stream, ensuring that any buffered characters are written to the file.

c) close(): Closes the output stream.

Example:

```java
FileWriter fw = new FileWriter("example.txt");
fw.write("Hello, World!"); // Writes the string to the file
fw.close();
```

8. BufferedReader:

Definition: The BufferedReader class reads text from a character input stream, buffering the characters for efficient reading of characters, arrays, and lines.

Methods:

a) readLine(): Reads a line of text from the input stream.

b) read(char[] cbuf, int off, int len): Reads characters into an array from the input stream.

c) skip(long n): Skips over and discards n characters from the input stream.

d) close(): Closes the input stream.

Example:

```java
FileReader fr = new FileReader("example.txt");
BufferedReader br = new BufferedReader(fr);
String line = br.readLine();
System.out.println(line); // Output: "Hello, World!"
br.close();
```

Note: The examples provided assume that the file "example.txt" exists and is accessible for reading and writing.

5.1.3 Reading and writing text files.

Sure! I'll explain the concept of reading and writing text files and provide examples of four methods commonly used in this process. Please note that the response will be split into multiple messages due to the character limit. Let's get started!

Reading and writing text files involve manipulating files that contain textual data. This operation is commonly performed in programming to read data from external files or write data to them. In Python, there are several methods available to achieve these tasks efficiently.

1. Reading Text Files:

Reading a text file involves opening the file, reading its contents, and then closing the file. The file's contents can be read as a whole or line by line.

Example:

Let's assume we have a file named "example.txt" with the following content:

```
Hello, World!
How are you?
```

```python
# Opening and reading a text file
file = open("example.txt", "r")
content = file.read()
file.close()

# Output
print(content)
```

Output:

```
Hello, World!
How are you?
```

2. Writing Text Files:

Writing to a text file involves creating a new file or overwriting an existing file, writing content to it, and then closing the file. The content can be written as a whole or line by line.

Example:

```python
# Opening and writing to a text file
file = open("example.txt", "w")
file.write("This is a new line.\n")
file.write("Another line.\n")
file.close()

# Output
# The file "example.txt" will be created/overwritten with the given content.
```

3. Appending Text Files:

Appending to a text file involves opening an existing file, moving the cursor to the end, writing new content, and then closing the file. The new content is added to the end of the file.

Example:

```python
# Opening and appending to a text file
file = open("example.txt", "a")
file.write("This is an appended line.\n")
file.close()

# Output
# The file "example.txt" will have the new content added to the end.
```

4. Reading Text File Line by Line:

Reading a text file line by line is useful when dealing with large files or when processing one line at a time. This method allows for efficient memory usage.

Example:

```python
# Opening and reading a text file line by line
file = open("example.txt", "r")
for line in file:
    print(line.strip())  # Strip removes the trailing newline character

file.close()

# Output
Hello, World!
How are you?
This is a new line.
Another line.
This is an appended line.
```

These methods provide a basic understanding of reading and writing text files in Python. Let me know if you need further clarification or have any other questions!

Sure! I'll explain the concepts of reading and writing text files, and then provide definitions for each concept along with four methods related to each. Here's the breakdown:

5.1.3 Reading and Writing Text Files

Reading and writing text files are fundamental operations in programming that allow you to interact with external files containing textual data. These operations are commonly used for tasks such as input/output, data storage, and data processing. Let's explore each concept in detail:

Definition: Reading Text Files

Reading a text file refers to the process of accessing and extracting data from an existing file. This data can be read and manipulated within a program for further processing or analysis.

Methods for Reading Text Files:

1. **`open()` function:** The `open()` function is used to open a text file in read mode, allowing you to access its contents. Here's an example:

```python
file = open("example.txt", "r")
```

2. **`read()` method:** The `read()` method reads the entire content of a text file and returns it as a string. Here's an example:

```python
file = open("example.txt", "r")
content = file.read()
print(content)
```

Output:

```
This is an example file.
It contains some text.
```

3. **`readline()` method:** The `readline()` method reads a single line from the text file and returns it as a string. It moves the file pointer to the next line after reading. Here's an example:

```python
file = open("example.txt", "r")
line1 = file.readline()
line2 = file.readline()
print(line1)
print(line2)
```

Output:

```
```

This is an example file.

It contains some text.

```

4. **`readlines()` method:** The `readlines()` method reads all the lines of a text file and returns them as a list of strings. Each element of the list represents a line from the file. Here's an example:

```python

file = open("example.txt", "r")

lines = file.readlines()

print(lines)

```

Output:

```

['This is an example file.\n', 'It contains some text.\n']

```

Definition: Writing Text Files

Writing to a text file refers to the process of creating a new file or overwriting the content of an existing file with new text-based data.
```

Methods for Writing Text Files:

1. **`open()` function:** The `open()` function is also used for writing text files. By specifying the "w" mode, the function opens the file in write mode, allowing you to write new content or overwrite the existing content. Here's an example:

```python
file = open("example.txt", "w")
```

2. **`write()` method:** The `write()` method writes a string to the text file. It can be called multiple times to write different strings or concatenate them. Here's an example:

```python
file = open("example.txt", "w")
file.write("This is the first line.\n")
file.write("This is the second line.")
file.close()
```

Output (in the "example.txt" file):

```
This is the first line.
This is the second line.
```

3. **`writelines()` method:** The `writelines()` method writes multiple lines to the text file. It

takes a list of strings as input, where each string represents a line. Here's an example:

```python
file = open("example.txt", "w")
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
file.writelines(lines)
file.close()
```

Output (in the "example.txt" file):

```
Line 1
Line 2
Line 3
```

4. **`append()` mode:** In addition to the "w" mode, you can use the "a" mode with the `open()` function to open a text file in append mode. This allows you to add new content to the end of an existing file without overwriting its previous contents. Here's an example:

```python
file = open("example.txt", "a")
file.write("This line will be appended.")
file.close()
```

Output (in the "example.txt" file):

```
Line 1

Line 2

Line 3

This line will be appended.
```

That covers the definitions of reading and writing text files, along with four methods related to each concept. Feel free to ask if you have any further questions!